Algebraic RAM

Pergament Evgenya
Algebraic RAM

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Pergament Evgenya

Submitted to the Senate of the Technion — Israel Institute of Technology
Kislev 5777 Haifa December 2016
This research was carried out under the supervision of Prof. Eli Ben-Sasson, in the
Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research
collaborators in conferences and journals during the course of the author’s master’s
research period, the most up-to-date versions of which being:

E. Ben-Sasson, I. Ben-Tov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Perge-
ment, M. Riabzev, M. Silberstein, E. Tromer and M. Virza, Computational
integrity with a public random string from quasi-linear PCPs, cryptology ePrint
Archive, Report 2016/646.

Acknowledgements

I would like to thank my advisor, Prof. Eli Ben-Sasson, for his excellent guidance,
patience and encouragement throughout my Master’s degree. It was my privilege to
work with Prof. Ben-Sasson.
I would also like to thank my parent for their endless love and support.

The generous financial help of the Technion is gratefully acknowledged.
Contents

List of Figures

Abstract 1

1 Introduction 3

2 Preliminaries 13
  2.1 Basic Algebra ............................................. 13
  2.2 Extension Fields and Closure .......................... 13
  2.3 Multivariate Polynomials ............................... 14

3 Computational Integrity 15
  3.1 Bounded Time RAM – Computational Model ........... 16
  3.2 Verification of Computational Integrity in Bounded Time RAM Model . 19
  3.3 The Transition Function as A Sequential Program .... 21

4 Reduction from $L_{CI}$ to $ACIP$ 23
  4.1 Modular Algebraic Construction ......................... 24
  4.2 Selector Polynomial ..................................... 27
  4.3 Decreasing the Number of Variables by Pre-selection ..... 29

5 $ACIP$ construction over $\mathbb{F}_{2^m}$ 33
  5.1 Type Checking ............................................. 35
  5.2 Integer Representation over $\mathbb{F}_{2^m}$ ................ 35
    5.2.1 Constraint System for Transition between Representation Methods 36
  5.3 ALU Opcodes .............................................. 39
    5.3.1 Auxiliary Opcodes .................................. 40
    5.3.2 Binary Opcodes ..................................... 46
    5.3.3 Integer opcodes .................................... 49
    5.3.4 Comparison Opcodes ................................. 58
    5.3.5 Jump Opcodes ........................................ 60
    5.3.6 Shift Opcodes ....................................... 62
  5.4 Memory validity ........................................... 64
List of Figures

1.1 Natural ALU implementation. ................................. 7
1.2 Our implementation. ........................................ 9
3.1 Bounded Time RAM module ................................. 17
4.1 A schematic description of how a sequential program evolves from one configuration to the next. ................................. 26
Abstract

With the increase of delegation of computation to powerful computational servers, questions of computational integrity and efficient verification start to arise. For instance, we may want to know if the execution was executed correctly or if the server had an incentive to misreport the result. How can the client who delegates a computation verify that the answer he received is the result of the computation? Probabilistically Checkable Proofs (PCP) can be used in order to verify the correctness of a computation that is executed by an untrusted server, such that the verification time is significantly less than the time needed to run the computation itself. They are, however, considered to be efficient only in theory. People believe that the constants that appear in the asymptotic measurements make PCPs impractical.

In this thesis we focus on arithmetization, i.e., encoding the transition function of a computation as an algebraic problem. Arithmetization is used in many theoretical and practical results, and is valuable in making PCPs practical. In order to encode the transition function, we define polynomials over a field $\mathbb{F}$ such that these polynomials are set to zero iff the computation corresponding to that function is executed correctly. The main contribution of our work is that we have significantly reduced the number of variables over which these polynomials are defined. This reduction improves both prover and verifier practical efficiency as well as the proof length. Also, ours is one of the first works, that uses arithmetization over a field of small characteristic. Our encoding uses a number of variables that is only linear in size of a register, even in fields of small characteristic.

Another main contribution of our work is a modular construction of polynomials for transition functions. We start by defining a set of polynomials for opcodes, such as \texttt{bitwise-XOR}, \texttt{ADD}, etc. These polynomials are set to zero iff an opcode is executed correctly. Then, we expand this set of polynomials to capture instructions. An instruction consists of an opcode and operands upon which the opcode operates. For example, \texttt{ADD R}$_1$, \texttt{R}$_2$, \texttt{R}$_3$, which denotes that register \texttt{R}$_1$, should be assigned with the addition result of registers \texttt{R}$_2$, \texttt{R}$_3$, namely, \texttt{R}$_1 = \texttt{R}$_2 + \texttt{R}$_3$. For each instruction, we define a set of polynomials that is set to zero iff an instruction is executed correctly. Finally, we expand these sets of polynomials to capture the transition’s function correctness. Using this encoding technique, one can arithmetize numerous programs, in particular any sequential code written in assembly language.
We implemented the arithmetization in C++ for assembly language based on TinyRAM assembly language. TinyRAM is a Random Access Machine (RAM) whose assembly language is concise yet, at the same time, is expressive and contains widely-used opcodes that produce efficient assembly codes for high-level languages. We present concrete measurements of the arithmetization for programs written in this assembly.
Chapter 1

Introduction

Motivation  The number of companies and individuals that outsource their computation to distant powerful servers is increasing dramatically every year. This phenomenon is called cloud computing, and it introduces many advantages for individuals and companies. To name a few, they can avoid the costs of buying powerful physical computers, replacing and maintaining them. Although, outsourcing a computation introduces many benefits, it raises a lot of questions about security, integrity and confidentiality due to the fact that an untrusted party is involved in executing the computation. A malicious distant server may benefit from not executing the program or from misreporting the correct answer. How can the client verify that the computation was in fact executed and that the result is correct?

Each computation can be described by an execution trace, a sequence in which the \(i^{th}\) entry holds all the machine’s values (e.g., register value) after \(i\) cycles. A naive way to verify that the program is executed correctly is by checking that each two consecutive entries in the execution trace comply with the program. This method of authentication, however, requires the same resources and time as the execution of the computation itself. The goal is to outsource a computation to an untrusted server such that the client would be able to verify that the program is executed correctly in significantly less time than the time required to run the computation.

PCP  One can describe the problem above in a more mathematical way by defining a language, \(L = \{(a,b) | f(a) = b\}\), such that \(f\) is the computation the client wants to execute and \(a\) is the input. The server, in addition to sending the result \(b\), also sends a proof that confirms that \(f(a) = b\). One way to generate proofs such that the client is convinced that \(f(a) = b\) in much less time than the computation itself is with probabilistically checkable proofs (PCP) [BFL90, BFLS91, AS98, ALM+98]. Generally speaking, a PCP system for a language \(L\) consists of a probabilistic verifier (client) and a proof \(\pi\). The verifier on an input \(x = (a,b)\) should decide whether the input belongs to the language, \(L\), by reading only few bits of the proof. If \(x \in L\), then there exists a proof \(\pi\) that the verifier accepts. Otherwise \((x \notin L)\) and the verifier rejects with probability
of at least $\frac{1}{2}$ all the proofs, $\pi^*$. The primary constructions of PCP proofs were of length $\text{poly}(T)$, where $T$ is the length of the execution trace [HS00, BSGH'06]. Today, the state-of-the-art PCP proofs are quasi-linear time in $T$, [BS08, Din07, BGH+05, Mie09], and can be computed in quasi-linear time in $T$ [BSCGT13]. Although PCPs are considered to be asymptotically efficient, people tend to believe that these kinds of proofs are impractical. The common belief is that the constants that “hide” in the asymptotic measurements are very large, and hence, both the prover’s and verifier’s execution time would be too long for practical purposes [WB15, SMBW12].

**From theory to practice** This thesis is part of a larger project that shows a practical reduction from an assembly language to a PCP proof, and hence, proves that PCP proofs can be used in practice. In this thesis, we focus on the first part of the reduction, encoding the transition function as an algebraic problem. A transition function is a well-known concept describing how a computation evolves from a current machine state to the next state. One can also think of the transition function as a relation, such that two machine states belong to the relation iff they were, in fact, two consecutive states during the execution of the computation. We translate the problem of checking if two machine states belong to the relation induced by the transition function into a problem that checks whether a set of polynomials is set to zero over a field $\mathbb{F}$. Such an encoding is also called an arithmetization, and it is a key component in many theoretical and practical results. The key idea behind arithmetization is that the polynomials can be assigned with myriad assignments over a field, and not only with the possible machine state values. An efficient arithmetization is of major importance in making PCP’s practical, because it has a direct effect on both the prover’s and verifier’s running time and efficiency in practice.

**Our work** We assume that the prover provides the verifier with an execution trace, $S$. If the program is executed by an honest prover, each two consecutive entries in $S$ belong to the relation induced by the transition function. In our work, we define a set of polynomials over a field $\mathbb{F}$ that “captures” this relation. Namely, these polynomials are set to zero if and only if each two consecutive entries in the execution trace belong to the transition’s function relation. Checking that each two consecutive entries belong to the relation induced by the transition function, however, does not “capture” memory validity. In other words, checking the above condition does not guarantee that a value that was read from address $Addr$ in cycle $i$ was in fact written to the same address in the previous memory write access to address, $Addr$. In order to check memory validity, the prover provides the verifier with another trace $S_{\text{memory\,sorted}}$, a permutation of $S$ sorted first by memory address and then by location in $S$. We say that $S_{\text{memory\,sorted}}$ is memory valid if for each two consecutive entries $i − 1$ and $i$, which access the same address $Addr$, the value that was read from $Addr$ in entry $i$, is the same value which was either written or read from $Addr$ in entry $i − 1$. Given a $S_{\text{memory\,sorted}}$, there is a
natural relation over consecutive pairs that captures the memory validity, and we can define a set of polynomials over a field $F$ such that these polynomials are set to zero iff each two consecutive entries in $S_{memory_{sorted}}$ belong to this relation.

We call a set of polynomials that is set to zero iff a relation, $R$, holds an algebraic constraint integrity problem (ACIP) for $R$. Given a program $P$, which accesses the memory unit, we define an ACIP for the program as a union of two ACIPs. The first ACIP is set to zero iff each two consecutive entries in $S_{memory_{sorted}}$ belong to the memory validity relationship and the second ACIP is set to zero iff each two consecutive entries in $S$ belong to the relationship induced by the transition function. Later, when discussing Modular Construction, we explain how the second ACIP can be defined in a modular way such that it is the union of ACIPs for sub-relations. Note that if a program $P$ does not access the memory, then there is no need to define an ACIP that checks memory validity.

We define our ACIP over three disjoint sets of variables. Two of these sets represent two consecutive entries in a trace, either in $S$ or in $S_{memory_{sorted}}$. Although, any relation can be defined by polynomials over these two sets of variables, the degree of such polynomials may be high. Therefore, we define a third set of variables, which we call auxiliary variables, that help us reduce the degree of these polynomials. For example, denote that the variables $X_1, X_2, \ldots, X_n$ belong to the first two sets and the polynomial $p$ is defined over these variables such that $p(X_1, X_2, \ldots, X_n) = X_1 \cdot X_2 \cdot \ldots \cdot X_n$. The degree of this polynomial is $n$ and it can be reduced to 2 by adding $n - 2$ auxiliary variables and $n - 1$ polynomials

$$p_1(X_1, X_2, Y_1) = Y_1 + X_1 \cdot X_2$$
$$p_2(X_3, Y_2) = Y_2 + Y_1 \cdot X_3$$
$$\vdots$$
$$p_{n-1}(X_n, Y_{n-1}) = Y_{n-2} \cdot X_n$$

There are three parameters in this work that we take into consideration when defining an ACIP: (1) the number of variables needed to define the ACIP; (2) the maximal degree of a polynomial in the ACIP; and (3) the number of polynomials in the ACIP. Each parameter has an effect on both the prover and verifier execution time and the length of the PCP proof. Therefore, decreasing these parameters is a step towards making PCPs practical.

To simplify the discussion in this chapter, we assume that the executed program is given as a sequence of instructions $P = \{I_0, \ldots, I_{\ell-1}\}$. Each instruction commands the processor to execute an opcode such as bitwise-AND or ADD, on operands that are either constants, memory locations or registers. The program is executed on a machine that has a memory unit and a finite number of registers. One special register is called the program counter (PC). It can be assigned with a value in $\{0, \ldots, \ell - 1\}$, which
indicates the instruction which is executed. In our work, we refer to the *arithmetic logic unit* (ALU) as a modular component that comprises all the opcodes’ polynomials (see next paragraph for more details), rather than a hardware circuit that performs logical and arithmetic operations. In the same way, when we discuss circuits in this work, we refer to them as components that can easily be turned into polynomials over a field rather than a low-level hardware. A straightforward way to translate a circuit into a set of polynomials is by defining a variable for each gate in the circuit, and assigning either a gate’s output or a circuit’s input to the wires. Then, one can define a polynomial for each gate, over the gate’s inputs and gate’s variable, which mimics the gate’s operation; the polynomial is set to zero iff the values are valid for the gate. For example, given a binary *AND* gate, whose inputs are denoted \( X_1 \) and \( X_2 \), we define a variable \( X_{\text{and}} \), and a polynomial \( X_{\text{and}} = X_1 \cdot X_2 \). If \( X_1 \) and \( X_2 \) are assigned only binary values, then \( X_{\text{and}} \) is set to 1 iff both \( X_1 \) and \( X_2 \) are assigned as 1, which matches the behavior of a binary *AND* gate. One can use the arithmetization techniques presented in [BF91], in order to encode the whole circuit.

**Contribution** This work makes three key contributions: (1) decreasing the number of variables by pre-selection; (2) algebraic arithmetization over small characteristic fields; and (3) modular construction. We elaborate each point below. We believe that decreasing the number of variables by pre-selection is the most novel and important contribution of these three. That said, we first discuss our modular construction in order to describe notions and terms that will be used later.

**Modular construction** Given a program \( P \) of length \( \ell \), the construction of the program’s ACIP is done in a modular way. It is built bottom-up and has two main steps. In the first step, we define an ACIP for an opcode. In the second step, we expand the opcode’s ACIP to an ACIP for an instruction. The program’s ACIP is the union of all the instructions ACIPs.

- **opcode** We start by defining an ACIP for opcodes. An opcode can be seen as a relation between \( k \) inputs and \( l \) outputs. For example, an addition of two numbers can be viewed as a relation between \( k = 2 \) inputs and \( l = 1 \) outputs. All triples \((a, b, c)\) such that \( a + b = c \) belong to this relation. As a first step, we define an ACIP for numerous opcodes. An ACIP for an opcode is satisfied, all the polynomials are set to zero, if and only if a \( k + l \) tuple belongs to the relation induced by the opcode.

- **instruction** The second step of the modular construction is to define an ACIP for an instruction. As stated above, an instruction consists of an opcode and operands. An example for an instruction is \( \text{ADD } R_1, R_2, R_3 \), which denotes that register \( R_1 \) should be assigned the addition results of registers \( R_2 \) and \( R_3 \), namely, \( R_1 \leftarrow R_2 + R_3 \). The ACIP we define for an instruction consists of the opcode’s
constraints (polynomials) and the polynomials that are set to zero iff the opcode’s inputs and outputs are wired with the operands given in the instruction. In the example above, the opcode’s inputs should be assigned the values \( R_2 \) and \( R_3 \) hold, and \( R_1 \) should be assigned the opcode’s output. We add to this ACIP another two types of polynomials. The first type is set to zero if and only if the registers that are not modified by the instruction have the same value before and after the instruction’s execution. For instance, in the presented example, the values of \( R_2 \) and \( R_3 \) should not be modified during the execution of the instruction. The second type of polynomials are set to zero if the PC register is modified correctly. In most instructions, the PC should be increased by one. There are instructions, however, such as jump, that consist of opcodes that may change the PC to point to other values in \( \{0, \ldots, \ell - 1\} \).

**Decreasing the number of variables by pre-selection** We now describe the most important contribution of this work: the reduction of the total number of variables in the system. Recall, that here the ACIP’s polynomials are defined over three disjoint sets of variables, and the total number of variables used to define the program’s ACIP has an impact on the practicality of the PCPs. Therefore, keeping the total number of variables low is critical.

The most natural way to construct the ALU would be by defining circuits for individual opcodes (which are later translated into polynomials), and then use multiplexers to choose the correct output based ultimately on the PC value. We call this method *post-selection* because all opcodes are computed in parallel, and only afterwards does the multiplexer chooses one result out of numerous options. Figure 1.1 illustrates this solution.

![Figure 1.1: Natural ALU implementation.](image)

Let \( n_{ops} \) denote the number of different opcodes the machine specification denote. \( op_1, \ldots, op_{n_{ops}} \)'s polynomials are defined over a unique set of variables. When all the polynomials are defined, a multiplexer chooses one result from \( n_{ops} \) options, according to the multiplexer’s inputs, which comes from the PC register.
The number of variables that each opcode uses is at least the number of gates in that opcode’s circuit. Therefore, the total number of variables in the system is at least the summation of the number of variables that each opcode uses, plus the multiplexer circuit size.

\[ \#\text{variables} = \text{Circuit Size} = \sum_{i=1}^{n_{\text{ops}}} (\text{op}\_\text{variables}(i)) + O(n_{\text{ops}}), \]

where \( n_{\text{ops}} \) is the number of opcodes and \( \text{op}\_\text{variables}(i) \) is the number of variables used by the \( i \)th opcode. In our novel solution we use pre-selection in order to decrease the number of variables significantly. The main observation is that during each cycle, only one opcode is executed. Given the value of the PC, we know exactly which instruction and opcode should be executed. We call this method pre-selection, because we choose which opcode should be executed by the PC values instead of computing the result of all the opcodes. Given a program \( P = \{I_1, \ldots, I_\ell\} \), we define a global set of variables. This set of variables is equal in size to the maximum variable set size an opcode in a program requires. For example, if the program consists of \( n \) different opcodes such that these opcodes require \( \{nv_1, nv_2, \ldots, nv_n\} \) variables, then the global set we define is of size \( \max \{nv_1, nv_2, \ldots, nv_n\} \). All opcodes’ ACIPs are defined over this global set of variables. After defining an ACIP for each opcode in the program, we multiply these polynomials by an appropriate selector polynomial. A selector polynomial for the \( i \)th instruction is a polynomial that is defined over the PC variables, such that it is set to a non-zero value when the PC is assigned \( i \) and is 0 for the value \( \{0, \ldots, \ell - 1\} \setminus \{i\} \). The multiplication by the selector polynomial sets to zero polynomials that are not needed for the opcode currently executed. While increasing the degree slightly, by \( \lceil \log(\ell) \rceil \) (the degree of a selector polynomial), using our method leads to a significantly smaller number of variables, namely,

\[ \#\text{variables} = \max_{i=1}^{n_{\text{ops}}} (\text{op}\_\text{variables}(i)). \]

Clearly:

\[ \sum_{i=1}^{n_{\text{ops}}} (\text{op}\_\text{variables}(i)) + O(n_{\text{ops}}) \geq \max_{i=1}^{n_{\text{ops}}} (\text{op}\_\text{variables}(i)) \]

In both implementations when a program uses only a subset of the opcodes, there is no need to define ACIPs for opcodes that do not appear in the program. In addition to reducing the number of variables, our method minimizes the total number of polynomials in the system. Many variables in numerous opcodes have the same “role” – they can be assigned with the same set of elements, e.g., \( \{0, 1\} \). When there is only one global set of variables, one should only define one set of polynomials that checks these assignments; however, when there is a different set of variables for each opcode, these polynomials
should be applied on each relevant copy. In the last (Results) paragraph, we present concrete numbers that show the superiority of our construction.

Figure 1.2 describes the way we implement the ALU.

![Figure 1.2: Our implementation.](image)

Let \( n_{\text{ops}} \) denote the number of different opcodes in the machine specification. In this figure, each rectangle represents an opcode's ACIP. The program uses only a subset of the possible opcodes. Their ACIPs are multiplied by the appropriate selector polynomials, \( \text{Selector}(i) \) for \( i \in \{1, 2, 3\} \). The rectangles that are crossed represent opcodes that do not appear in the program.

**Algebraic arithmetization over small characteristic fields** The state-of-the-art work on PCP construction for RAM programs [BSCGT13] encodes the computation as a Boolean circuit. Although any computation can be presented as a Boolean circuit and such an implementation is feasible, it lacks any algebraic flavor. For example, the implementation of an XOR operation is more complex via a Boolean circuit than over a field of characteristic-2. In a Boolean circuit, the numbers \( \text{num}_1 \) and \( \text{num}_2 \) are viewed as a vector of bits such that \( \text{num}_1 = a_n, \ldots, a_0 \) and \( \text{num}_2 = b_n, \ldots, b_0 \) where \( a_i, b_i \) are the \( i \)-th bit of \( \text{num}_1 \) and \( \text{num}_2 \) respectively, and the XOR on corresponding bits is executed with 2 AND gates, 2 NOT gates and an OR gate \((a_i \cdot \bar{b}_i + \bar{a}_i \cdot b_i)\). Addition over an algebraic field of characteristic-2 is essentially equal to XOR. The definition of polynomials over a field can be more succinct such as in the XOR case. Moreover, the reduction from the ACIP goes through an algebraic constraint satisfaction problem (ACSP) [BSCGT13, CZ15, BS08], which is an algebraic problem. Hence, it is more natural and efficient to define the polynomials over a field while using the field’s properties. Ours is one of the first works to present a concrete reduction from an assembly language to algebraic problems over fields of small characteristics. Fields of small characteristics present special difficulties, especially for integer based opcodes such as multiplication and comparison; in prime fields that are sufficiently large, such issues are easier to handle because field addition and multiplication behave like integer addition and multiplication (for sufficiently small integers). We present a way to encode
many widely-used opcodes as an algebraic problem over characteristic-2. Translating some opcodes such as AND or NOT to ACIP is intuitive. The encoding of other opcodes such as multiplication and comparison is more challenging. One should always keep in mind that the objective is to minimize the number of polynomials, their degree and the number of auxiliary variables. The naive way to implement these opcodes is not always the most advantageous, and can be considered wasteful. Our reduction requires only a constant degree and the number of auxiliary variables that is linear in the register bit size, denoted by $W$. In particular, the comparison opcode requires $3 \cdot W$ auxiliary variables and the multiplication opcode requires only $8 \cdot W$ auxiliary variables.

**Prior Work in Other Models** Arithmetization is a pivotal technique used in numerous theoretical and practical results on which we now elaborate. Many works tend to view a computation as an Boolean circuit. Goldwasser, Kalai and Rothblum (GKR) [GKR15] focused on proofs with an efficient (polynomial) prover and a super-efficient (linear) verifier. In their work, the starting point is a uniform family of boolean circuits. In GKR, they used arithmetization in order to validate the computation of the circuit by using the sum-check protocol. Understanding the importance of interactive proofs (IP) [GMR85, Bab85] in practice, several works, such as [CMT12, Tha13, TRMP12] refined GKR’s work and provided a practical implementation.

Another interesting line of research which is based on [IKO07], moves a step closer towards making verification of general-purpose computations a practical problem. These projects address the problem of efficient encoding, by mapping the computation to equivalent constraints over a finite field (arithmetic circuits), instead of the boolean circuits. Ginger [SVP+12], which is based on Pepper [SMBW12], transformed broad class of computations into degree-2 polynomials over a field $F$. Zaatar [SBV+13] imposed another requirement on the constraints, such that the constraints are of form $A \cdot B = C$ where $A, B, C$ are polynomials of degree 1. Another type of work that used rank-1 quadratic constraints are Succinct Non-interactive ARgument of Knowledge (SNARKs), based on Quadratic Span Problem (QSP) [BCG+13, GGPR13]. The two main differences between our work and these are: (1) we use fields of small characteristics to encode the computation while they use prime fields of large characteristics and (2) the kind of constraints (gates) used. In our work, we allow polynomials of degrees higher that 2, instead of constraints of rank-1 quadratic forms. The use of constraints of rank-1 quadratic forms makes it difficult to implement our novel solution, reuse of variables and pre-selection. In our work, we significantly decrease the number of variables in the system, by increasing slightly the degree. Multiplying a bilinear form by a single variable already makes it a cubic, non-quadratic, form. We leave to future work to find a simple way to apply our solution when the constraints are restricted to rank-1 quadratic form.

The state-of-the-art work [BSCGT13], which is the similar to our over-all goal, encodes the computation as a boolean circuit. In SNARK [BCG+13], they referred to
[BSCGT13] and stated that although such an encoding is asymptotically efficient, it is quite challenging, in practice, to define such a circuit to be of a manageable size. Therefore, they obtain an efficient arithmetization by using arithmetic in prime fields of large characteristic. In our work, we improve the encoding of computation over fields of small characteristics, such as $\mathbb{F}_{2^m}$.

**Results** The subset-sum is an important problem in computation theory. Given a multiset $A$, and an integer $n$, one should decide whether there exists a subset of $A$ whose sum is $n$. The exhaustive subset-sum program presented in [BBC+16] checks, exhaustively, all the possible subsets, $2^{|A|}$ in number, in order to solve the subset-sum problem. This program is written in 16 bit TinyRAM assembly language. TinyRAM is a random access machine whose assembly language is, on the one hand, concise, and on the other hand, consists of widely-used opcodes and is expressive for writing succinct programs. The length of the program is 22 instructions. Hence in our work, the PC register is represented by five variables ($\lceil \log(22) \rceil$) in the first two sets of variables discussed in the beginning of this chapter. The total number of variables that represent an entry in the execution trace is 22, and includes variables that represent the PC, flag and registers. Our $ACI$, for the exhaustive subset-sum, is defined over 114 variables, such that 44 are used to represent two consecutive entries in the execution trace, and there are 70 auxiliary variables. This number clearly makes sense, because the $ADD$ opcode (used in this program) uses $4 \cdot W - 1 = 4 \cdot 16 - 1 = 63$ variables. The other variables are used for general purpose such as ALU’s input and output variables. Moreover, there are only 85 polynomials in the system, and the upper bound on the highest degree is 12. These numbers are a tremendous improvement over the natural way implementation. For example, if we defined the constraints in the natural way discussed above, we would have at least $\times 2.1$ auxiliary variables and polynomials, while the maximum degree will be reduced to 7 (see Chapters 4 and 5 for more details). These numbers escalate when a program comprises all the opcodes in some assembly language, and not only ones from a subset as occurs in the exhaustive subset-sum program. For example, there are 27 different opcodes in the TinyRAM assembly language, and each opcode’s $ACIP$ requires on average, at least, $2.6 \cdot W$ (Table 5) variables. Therefore, there would be over 1000 variables in the system, and around the same number of polynomials.
Chapter 2

Preliminaries

This chapter provides a general background on groups, fields, and polynomials.

2.1 Basic Algebra

Our primary interest in this thesis is finite fields; a field with a finite number of elements. For a finite field $\mathbb{F}$, there is a prime number $p$ and $m \geq 1$ such that $|\mathbb{F}| = p^m$, and for every prime order there is a finite field of this order. Finite fields are usually called Galois fields and are denoted as $\mathbb{F}_{p^m}$ or $GF(p^m)$. In this work, our main focus is fields of characteristic 2, $\mathbb{F}_{2^m}$ or $GF(2^m)$.

The set of polynomials with coefficients in $\mathbb{F}$ is denoted by $\mathbb{F}[X]$. An irreducible polynomial over $\mathbb{F}$ is a non-constant polynomial that cannot be factored into two non-constant polynomials. There is an irreducible polynomial of degree $m \geq 1$ for every prime $p$ in $\mathbb{F}_p[X]$. If $I(X) \in \mathbb{F}_p[X]$ is an irreducible polynomial of degree $m$, then $\mathbb{F}_p[X]/I(X)$ is a field of size $p^m$ and $\mathbb{F}_p[X]/I(X) \simeq \mathbb{F}_{p^m}$. This implies that we can represent elements in $\mathbb{F}_{p^m}$ as polynomials of a degree strictly less than $m$.

Let $\mathbb{F}_{p^m}$ be a finite field. The multiplicative group $\mathbb{F}_{p^m}^* \equiv \mathbb{F}_{p^m} \setminus \{0\}$ is cyclic. There exists a generator $g$ that generates the whole group

$$\mathbb{F}_{p^m}^* = \langle g \rangle = \langle g^n \mid n \in \mathbb{Z} \rangle$$

2.2 Extension Fields and Closure

Let $\overline{\mathbb{F}}$ and $\mathbb{F}$ be fields. We say that $\overline{\mathbb{F}}$ is an extension field of $\mathbb{F}$ if $\mathbb{F} \subseteq \overline{\mathbb{F}}$ and $\mathbb{F}$ is closed under the operations of $\overline{\mathbb{F}}$. For example, $\mathbb{F}_{p^m}$ for $m \geq 1$ is a field extension of $GF(p)$.

A field $\overline{\mathbb{F}}$ is said to be algebraically closed if it contains a root for every non-constant polynomial in $\mathbb{F}[X]$. We say that $\overline{\mathbb{F}}$ is the algebraic closure of $\mathbb{F}$ if it is an extension field of $\mathbb{F}$ and algebraically closed.
2.3 Multivariate Polynomials

A multivariate polynomial $P$ in $X_1, X_2, \ldots, X_m$ over $\mathbb{F}$ is defined by

$$P = \sum_i \alpha \cdot X_1^{i_1} \cdot X_2^{i_2} \cdot \ldots \cdot X_m^{i_m},$$

where $i_j \in \mathbb{N}$ for $j = 1, \ldots, m$ and $\alpha \in \mathbb{F}$. The set of polynomials over the variables $X_1, X_2, \ldots, X_m$ in $\mathbb{F}$ is a ring and denoted by $\mathbb{F}[X_1, \ldots, X_m]$. The degree of $p_i$ is

$$\deg(p_i) = \sum_{j=1}^{m} i_j,$$

and the degree of the polynomial is defined by

$$\deg(P) = \max \{p_i | p_i \in P\}.$$

For $\mathcal{P} = \{P_1, \ldots, P_n\} \subset \mathbb{F}[X_1, \ldots, X_m]$, define the algebraic set of $\mathcal{P}$ to be its set of common roots,

$$V(\mathcal{P}) = \{(x_1, \ldots, x_m) \in \overline{\mathbb{F}}^m | P_1(x_1, \ldots, x_m) = \ldots = P_n(x_1, \ldots, x_m) = 0\},$$

where $\overline{\mathbb{F}}$ is the algebraic closure of $\mathbb{F}$. We are mostly interested in the set of roots that lie in $\mathbb{F}$ and denote it by $V_F(\mathcal{P}) = V(\mathcal{P}) \cap \mathbb{F}^m$. The degree of $\mathcal{P}$, denoted $\deg(\mathcal{P})$, is the maximal degree of an element of $\mathcal{P}$ and its algebraic complexity denoted $C(\mathcal{P})$ is the sum of sizes of arithmetic circuits computing $P_1, \ldots, P_n$. 

14
Chapter 3

Computational Integrity

In this chapter we discuss what is involved in verifying the correctness of an outsourced computation executed on a Random Access Machine (RAM). The key focus of this chapter is to explain how to verify the correctness of a computation in a purely combinatorial world. Later in this thesis we translate these concepts into an algebraic world. We are specifically interested in computations that are executed for a finite number, $T$, of computational cycles. In Section 3.1, we define the Bounded Time RAM (BT-RAM) model, which runs general programs for a finite number of cycles. We use the BT-RAM model because it is the most natural starting point. This model resembles widely-used computers, and with the help of modern compilers, the translation from a high-level language (such as C or Java) into an assembly language, and particularly into the BT-RAM model as defined in Section 3.3, is straightforward.

An instance of the computational integrity language ($L_{CI}$) is of the form $(M, B)$, such that $M$ is a BT-RAM and $B$ is a set of boundary constraints. Informally, boundary constraints indicate specific values that a machine state must respect during the execution of the program. An example for a boundary constraint is the demand that the initial state is $q_0$, as required in many computational models such as a Turing Machine and a Deterministic Finite Automaton. In Section 3.2 we elaborate on what is needed in order to determine whether an instance belongs to $L_{CI}$. The essence of this section lies in checking the correctness of an execution trace. An execution trace is a sequence of $T$ configurations such that the $i^{th}$ configuration describes the machine’s state (e.g., register values) after the $i^{th}$ execution cycle ends. Informally, we say that an instance $(M, B)$ belongs to $L_{CI}$, if there exists an execution trace that describes an actual computation of $M$ and it complies with the boundary constraints. One can be convinced that an execution trace indeed represents a correct computation if it is time and memory valid and consistent with the boundary constraints. Informally, we say that an execution trace is time valid if each two consecutive configurations represent two consecutive machine states that conform with the program. It is memory valid if the value read from some memory location is the exact value that was previously written to this location. An execution trace is consistent with the boundary constraints, if the specific values stated
in these constraints match the values that appear in the execution trace.

We now present a formal definition of the computational integrity language, which is based on the informal definition and concepts we described above. Through the chapter we focus on and describe each concept formally. We also explain why they are crucial in verifying the correctness of a computation. One can find a formal definition of

1. BT-RAM in Definition 3.1.1
2. Configuration and execution trace in Definition 3.1.2
3. Boundary constraints in Definition 3.2.4
4. Memory and time validity in Definition 3.2.2

**Definition 3.0.1** (Computational Integrity language). An instance of the computational integrity language is a tuple \((M, B)\) where \(M\) is a bounded time RAM machine and \(B\) a set of boundary constraints. The computational integrity language, \(L_{CI}\), is a set of instances for which there exists an execution trace which conforms with \(M\) that is memory and time valid, and that is consistent with \(B\).

This chapter has three sections. The first section, computational model (3.1), describes the finite computational model we use in our work. The second section, (3.2), elaborates on the exact conditions one should check in order to be convinced that an execution was executed correctly in our model. In other words, in this chapter we discuss what is involved in determining if a \((M, B)\) is an instance in \(L_{CI}\). Finally, in (3.3) we focus on a specific type of transition functions, represented by sequential programs, which resemble widely-used assembly languages.

### 3.1 Bounded Time RAM – Computational Model

In this section we define a bounded time Random Access Machine (BT-RAM), which is a computational model that runs for a finite number, \(T\), of steps. This model is essentially equal to the RAM machine \([CR72]\) for a program that executes a finite number of cycles.

**Definition 3.1.1** (The Bounded Time RAM). A **Bounded Time RAM** \(M\) is a tuple \(M = (T, R, Q, \Sigma_i, \Sigma_r, \Gamma_v, \Gamma_a, \delta)\) where

- \(T\) is a binary representation of a positive integer signifying the number of timesteps the machine executes.
- \(R\) is a finite set of registers. A register is a discrete memory location that can hold a finite number of bits.
- \(Q\) is a finite set of states.
- $\Sigma_i$ – is a finite input alphabet. It contains a special “no-input” symbol $\epsilon$.

- $\Sigma_r$ – is a finite register content alphabet.

- $\Gamma_v$ - is a finite memory content alphabet. It contains a special symbol called $load$, denoted $\gamma_\ell$, which will signal a memory read request.

- $\Gamma_a$ – is a finite memory address space alphabet. It contains a special null address symbol $\perp$.

- $\delta$ – is a transition function $\delta : Q \times \Sigma^R_r \times \Gamma_a \times \Gamma_v \times \Sigma_i \rightarrow Q \times \Sigma^R_r \times \Gamma_a \times \Gamma_v$.

**Note.** For simplicity, later in this thesis we assume that all the alphabets are equal.

The BT-RAM communicates with an external memory-module (see Figure 3.1). The memory module is a finite space of memory that is partitioned into cells, such that each cell has an address. We denote $\Gamma_a$ and $\Gamma_v$ as a memory address space alphabet and a memory content alphabet, respectively. In each communication with the memory module, BT-RAM sends two values $Addr \in \Gamma_a$ and $Value \in \Gamma_v$. There are three combinations of interest: (1) $Addr = \perp$, which indicates that the machine does not access the memory module; (2) $Addr \neq \perp \wedge Value = \gamma_\ell$; which indicates that the BT-RAM requires to read the value written in a cell whose address is $Addr$. We denote such a command as a $load$ command. (3) $Addr \neq \perp \wedge Value \neq \gamma_\ell$; which indicates that the memory module should write a $Value$ in a cell whose address is $Addr$. We denote such a command as a $store$ command. The transition function $\delta$ takes, as an input, a value and address that arrives from memory and outputs a memory address and a value to be written there. We denote by $\delta(C)|_{\Gamma_a}$ and $\delta(C)|_{\Gamma_v}$ the memory address and the value that the transition function outputs, respectively, such that $C \in Q \times \Sigma^R_r \times \Gamma_a \times \Gamma_v \times \Sigma_i$.

![Figure 3.1: Bounded Time RAM module](image-url)
Recall the NEXP-complete variant of the bounded halting problem:

\[ L_{BH} = \{(M, t) \mid \exists x, M \text{ (Turing machine) accepts } x, \text{ within } t \text{ steps, } t \text{ is given in a binary representation}\} \]

**Theorem 3.1.** \( L_{BH} \leq_p L_{CI} \)

**Proof.** We show a poly-time reduction that given an well-formed instance of \((M_{TM}, t) \in L_{BH}\), outputs a well-formed instance \((M_{BT-\text{RAM}}, B)\) of \(L_{BH}\), s.t. \((M_{TM}, t) \in L_{BH} \iff (M_{BT-\text{RAM}}, B) \in L_{BH}\).

Given \((M_{TM}, t) \in L_{BH}\), we define \(M_{BT-\text{RAM}}\) such that it has one register \(R\) that indicates the place of \(M_{TM}\)'s head. Each time the head moves to the right, we increase \(R\) by one, and when it moves to the left, we decrease \(R\) by one (if \(R\) equals zero, it stays zero). Our BT-RAM communicates with a memory module with \(t\) cells. \(\delta_{BT-\text{RAM}}\) simulates each step of \(\delta_{TM}\) in three steps: (i) it writes to the memory, then (ii) increases or decreases \(R\), and finally, (iii) reads from the cell whose address appears in \(R\). Because we simulate each step of \(\delta_{TM}\) by three steps \(\delta_{BT-\text{RAM}}\), we define \(T\) of \(M_{BT-\text{RAM}}\) to be \(3 \cdot t\) and \(|Q_{BT-\text{RAM}}| = 3 \cdot |Q_{TM}|\). When \(\delta_{TM}\) changes the current state from \(q_i\) to \(q_j\), \(\delta_{BT-\text{RAM}}\) changes the states from \(q_i^1 \rightarrow q_j^2\) when a memory store is executed, from \(q_i^2 \rightarrow q_j^3\) when \(R\) is modified according to the head’s movement, and from \(q_i^3 \rightarrow q_j^4\) when a memory load is executed. We add boundary constraints that check that the initial state is \(q_0^1\), and that the final state is \(q_f^1\). Clearly, the reduction we defined runs in polynomial time in the length of the input \((M_{TM}, t)\). Because our \(M_{BT-\text{RAM}}\) simulates the behavior of \(M_{TM}\), it is obvious that \((M_{TM}, t) \in L_{BH}\) if and only if \((M_{BT-\text{RAM}}, B) \in L_{BH}\). This completes the proof. \(\square\)

In order to capture the machine’s status (registers content, state, etc.) in a specific cycle during the execution of a program, we define a configuration. Informally, a configuration indicates state of the machine, the values all the registers hold, the input that is read, and the values the memory module sent. A sequence of \(T\) configurations is called an execution trace, and it describes the machine’s state step-by-step for \(T\) cycles.

**Definition 3.1.2** (Configuration, Execution trace). A configuration of \(M\) is a tuple \(C \in Q \times \Sigma_R^R \times \Gamma_a \times \Gamma_v \times \Sigma_i \times \Gamma_e\) where \(\Gamma_e = \{0, 1\}\). An execution trace of \(M\) is a sequence \(S = (C_0, \ldots, C_T)\) of configurations. The length of the execution trace is \(T\), and the width is the number of values each configuration holds, namely, \(|R| + 5\).

We call \(\Gamma_e\) a load flag, and each one of \(\{Q, \Sigma_R^R, \Gamma_a, \Gamma_v, \Sigma_i, \Gamma_e\}\) an entry in the configuration. Given a configuration \(C\) and an execution trace \(S\), we denote by \(C|_i\) the value that is written in the \(i\) entry in the configuration, e.g., \(C|_Q\) denotes the state that is indicated in the configuration. We say that \(C\)'s timestep is \(i\) if \(C\) is the \(i\)th configuration in \(S\), \(S[i] = C\). Later, in this thesis, we discuss augmented configurations that are composed of more values. The width of such configurations will be greater.
than $R + 5$. In the same way, we generalize the definition of an entry and the notion of $C_i$ when $i$ is an entry in the augmented configuration.

### 3.2 Verification of Computational Integrity in Bounded Time RAM Model

Recall that in Definition 3.0.1 and in the discussion above that definition we mentioned three things one should check in order to be convinced that an instance $(M, B)$ belongs to the computational integrity language $L_{CI}$. We said that $(M, B) \in L_{CI}$ if there exists an execution trace that is (i) time valid; (ii) memory valid; and consistent with the (iii) boundary constraints. We now examine each requirement. We define each condition formally, and explain how to check if a trace conforms to each one.

We “capture” the machine’s status after each cycle of the program’s execution by using configurations and an execution trace. Clearly, BT-RAM’s status is modified according to the transition function $\delta$. Each function can be viewed as a relation; therefore, we can define a relation $R_\delta$, which is induced by $\delta$. Informally, two configurations $C, C'$ belong to the relation $R_\delta$, if the transition function transforms between the matching machine states. Formally,

**Definition 3.2.1** (The relation induced by a transition function). Given a transition function $\delta$, and two configurations $C, C'$ we say that these configurations belong to the relation, $R_\delta$, induced by $\delta$ if:

1. The registers’ content, the machine’s state and the memory address are changed according to the transition function. Formally,
   $$C'|_{Q \times \Sigma^B \times \Gamma_a} = \delta(C)|_{Q \times \Sigma^B \times \Gamma_a}.$$

2. The memory value, which appears in $C'$’s configuration, matches the value written during a memory store command. Formally,
   $$\delta(C)|_{\Gamma_a} \neq \bot \land \delta(C)|_{\Gamma_v} \neq \gamma \ell \Rightarrow \delta(C)|_{\Gamma_v} = C'|_{\Gamma_v}.$$

3. The load flag, $\Gamma_\ell$, equals 1 if and only if a memory load command is executed. Formally,
   $$C'|_{\Gamma_\ell} = 1 \iff \delta(C)|_{\Gamma_a} \neq \bot \land \delta(C)|_{\Gamma_v} = \gamma \ell.$$

We say that an execution trace is time valid iff each two sequential configurations belong to the relation induced by $\delta$. It is memory valid if the value received from the memory module in a load command equals the value read or stored previously to the same address. Formally,

**Definition 3.2.2** (Time and memory trace validity). Let $S = (C_0, \ldots, C_T)$ be an execution trace as defined in Definition 3.1.2. $S$ is time valid if for every $i \in \{0, \ldots, T - 1\}$, two consecutive configurations belong to the relation induced by the transition function.
We say $S$ is memory valid if the following conditions hold for every $C_i$ with $C_i|\Gamma_a \neq \bot$ and $C_i|\Gamma_\ell = 1$ where $0 < i \leq T$. Let $i'$ be the maximal integer smaller than $i$ such that $C_i'|\Gamma_a = C_i|\Gamma_a$ then $C_i'|\Gamma_v = C_i|\Gamma_v$.

Recall from page 17 that a load command is executed if the BT-RAM sends the memory module two values, $Addr \in \Gamma_a$ and $Value \in \Gamma_v$, such that $Addr \neq \bot \land Value = \gamma_\ell$. In the same way, a store opcode is executed if $Addr \neq \bot \land Value \neq \gamma_\ell$.

Load commands are particularly problematic to verify because one must be sure that the exact value that was stored in the previous store command to the same memory address is read. These operations are the essence of memory validity. When checking whether the execution trace is memory valid, we verify that if some value was read in timestep $i$ from memory address $Addr$, the previous access to the same address $Addr$ either stored or loaded the same value. Given an execution trace $S$, such that each configuration contains $Addr \in \Gamma_a$, we can verify memory validity by sorting the trace lexicographically first by address, $\Gamma_a$ value, and then by timestep, and checking that each two consecutive configurations in the sorted trace, which access the same memory address, in which the latter configuration $\Gamma_\ell$’s value is 1, share the same value $\Gamma_v$. Formally, we can define memory validity by the formula

$$C_i'|\Gamma_a = C_i|\Gamma_a \neq \bot \land C_i'|\Gamma_\ell = 1 \Rightarrow C_i'|\Gamma_v = C_i|\Gamma_v,$$

where $C_i', C_{i+1}$ are two subsequent configurations in the sorted trace. We denote the sorted trace as $S_{memory\text{-}sorted}$ and assume that the configurations with $\bot$ value as $\Gamma_a$, are placed in the end of this trace. Although this decision is arbitrary (these configurations can be located at the beginning of the sorted trace as well), a decision must be made in order to enforce memory validity.

**Definition 3.2.3 (Memory sorted trace).** Given an execution trace $S$ (Definition 3.1.2), we say that $S_{memory\text{-}sorted}$ is a memory sorted trace if it is a permutation of $S$ and is sorted in ascending order first by address and then by timestep.

Thus far, the discussion focused on time and memory validity, which are basically a result of the program (or transition function). However, even if the execution trace is both memory and time valid, it can still represent a fallacious execution. For example, assume a malicious prover executes the program correctly but used the wrong input. In order to avoid such cases, the verifier should specify the value he expects some entry in a configuration to be. This type of constraint is called a boundary constraint and is essential to validate specific values.

**Definition 3.2.4 (Boundary constraints).** Let $S = (C_0, \ldots, C_T)$ be an execution trace. A boundary constraint on $S$ is a triple $B = (i, j, \alpha)$ where $i \in \{0, \ldots, T\}$ denotes a timestep, $j \in \{Q, \Sigma^R, \Gamma_a, \Gamma_v, \Sigma_i, \Gamma_\ell\}$ denotes an entry in the $i$th configuration and $\alpha$ is
its value. We say \( S \) is consistent with a set \( B = \{ B_1, \ldots, B_r \} \) of boundary constraints if for each \( 1 \leq i \leq r \), \( C_{B_i[0]}|_{B_i[1]} = B_i[2] \).

Special cases of boundary constraints are

- **initial configuration.** One may want to restrict the registers to be initialized to 0, and the initial state is some fixed state \( q_0 \).
- **final configuration.** Many theoretical computational models require the final state to be a \( q_{\text{accept}} \in Q \), which indicates the program was executed successfully.
- **explicit (deterministic) input.** In these boundary constraints, we want to check that if an explicit input is given by \( ((t^1, \sigma^1), \ldots, (t^n, \sigma^n)) \), then the \( \Sigma \) value in the configuration whose timestep is \( t^j \) equals to \( \sigma^j \) for each \( j = 1, \ldots, n \) namely, \( C_{t^j}[\Sigma_j] = \sigma^j \).

We assume that the prover sends us two execution traces \( S \), a time sorted trace, and \( S_{\text{memory sorted}} \), a memory sorted trace. The verifier’s objective is to check that these execution traces actually represent a valid program’s execution. We transform the problem of verifying that \( S \) is time and memory valid and consistent with the boundary constraints by verifying that:

1. **Time validity** – The trace \( S \) is time valid as defined in Definition 3.2.2.
2. **Memory validity** – The trace \( S_{\text{memory sorted}} \) is a memory valid as defined in Definition 3.2.2.
3. \( S_{\text{memory sorted}} \) is a permutation of \( S \) as required in Definition 3.2.3.
4. \( S \) is consistent **boundary constraints** as defined in Definition 3.2.4.

This thesis focuses on points (1) and (2), above. In the next chapters, we show how one can use algebraic methods in order to validate that \( S \) is time valid and \( S_{\text{memory sorted}} \) is memory valid. One can find a detailed discussion of (3) and (4) in other works such as \[ \text{BBC}^{+16}, \text{BSCGT}13, \text{CZ}15, \text{BS}08 \].

### 3.3 The Transition Function as A Sequential Program

So far \( \delta \) has been arbitrary. We now look at the special case of \( \delta \) conforming to a sequential program, or, simply a program. A program of length \( \ell \) is a sequence of instructions \( P = \{ I_0, \ldots, I_{\ell-1} \} \). An instruction consists of an opcode, which can be thought of as machine language operation such as bitwise-or or integer comparison, and of the relevant operands, which can be constants, register values, or memory location. Each instruction points to the next instruction in the program; by default, \( I_j \) points to \( I_{j+1} \) but certain instructions (such as jumps) may point to a different instruction,
and the pointer may further depend on the value of certain registers. In order for an
instruction to point to the next instruction, we assume the existence of a special register
called the program counter (PC). Given a program \( P = (I_0,\ldots,I_{\ell-1}) \), the PC can
hold values between \( \{0,\ldots,\ell-1\} \). This value indicates which instruction is going to
be executed. Each instruction modifies the PC value. By default, the PC is increased
by one but in special cases it can be modified to any point in the program.

An opcode \( op \) with \( k \) inputs and \( l \) outputs is defined by a relation that contains all
the sequences of inputs and outputs that correspond to a valid execution of \( op \). An
example is the addition opcode that adds two numbers and outputs the result, which
can be defined as an opcode with two inputs, \( k = 2 \), and one output, \( l = 1 \), and induces
the relation \( R_{\text{addition}} = \{(a,b,c) \mid a + b = c\} \).

**Definition 3.3.1 (Opcode).** Given an alphabet \( \Sigma \), a \((k, l)\) - opcode over \( \Sigma \) is a relation
\( R_{op} \subseteq \Sigma^{k+l} \) where \( l, k \geq 0 \). The first \( k \) elements are called inputs and the last \( l \) elements
are called outputs.

An instruction is a tuple (opcode, input-values, output-values) that connects an
opcode’s inputs and outputs and the operands that appear in the instruction. Moreover,
the instruction modifies the PC register. The PC register is always the first argument
in the input values and in the output values. As mentioned before, the default is to
increment the PC by 1, and later in the thesis, we often omit this fact when discussing
instructions. We shall discuss it only when we deviate from this standard (e.g., in
jumps).

**Definition 3.3.2.** Given an alphabet \( \Sigma \), a \((k+1, l+1)\)-instruction is a tuple \(((k, l) -
opcode, a_0, a_1, \ldots, a_k, b_0, b_1, \ldots, b_l)\) where

- \( \forall 1 \leq i \leq k : a_i \) is a register index or a value from \( \Sigma \).
  The values retrieved from \( a_1, \ldots, a_k \) are \((k, l)\) - opcode’s inputs.

- \( \forall 1 \leq i \leq l : b_i \) is a register index.
  The values assigned to \( b_1, \ldots, b_l \) are \((k, l)\) - opcode’s outputs.

- \( a_0 \) and \( b_0 \) are the PC register.

\( a_0, \ldots, a_k, b_0, \ldots, b_l \) are called the instruction’s arguments or operands, where the first
\( k + 1 \) arguments (operands) are called input arguments (operands), and the last \( l + 1 \) arguments (operands) are called output arguments (operands).

Given a program \( P = \{I_1, \ldots, I_{\ell}\} \), the transition function \( \delta \) defined by it is self-
evident. The BT-RAM machine has \( \ell \) states, and \( \delta \) transfers from the \( i \) th state to the \( j \) th
while executing the \( i \) th instruction on the machine state iff the \( i \) th instruction is followed
by the \( j \) th instruction. Going forward, we are going to assume that the transition is
specified as a program.
Chapter 4

Reduction from $L_{CI}$ to $ACIP$

In section 3 we discussed what one should check in order to determine whether an instance belongs to the computational integrity language $L_{CI}$ (Definition 3.0.1). As shown in Section 3.2, given two traces $S$ and $S_{memory\text{sorted}}$, there are four conditions one should check in order verify that a computation is executed correctly. In this work, we focus on two of them: (1) verifying that $S$ is a time valid trace and (2) verifying that $S_{memory\text{sorted}}$ is a memory valid trace. We translate the problem of verifying (1) and (2) into an algebraic world. We write a set of polynomials over a field $F$, such that these polynomials are set to zero iff: (i) each two consecutive configurations $C, C'$ in $S$ belong to the relation induced by the transition function (Definition 3.2.1) and (ii) each two sequential configurations $C, C'$ in $S_{memory\text{sorted}}$ are ordered according to the sorting order defined in Definition 3.2.3 and Equation 3.1 holds. If these polynomials are set to zero for each two consecutive configurations in $S$, we say that $S$ is a time valid trace. In the same way, we say that $S_{memory\text{sorted}}$ is memory valid, if these polynomials are set to zero for each two consecutive configurations in $S_{memory\text{sorted}}$. We may refer to polynomials that are defined over a field as algebraic constraints, or simply constraints. We say that a polynomial is satisfied by an assignment $\alpha$ in $F$ if $\alpha$ sets the polynomial to zero. Likewise, a constraint system (a set of polynomials) is satisfied by an assignment in $F$ if it sets all the polynomials to zero.

We define the constraints over three sets of disjoint variables. Two of these sets represent two sequential configurations. In each set, there is a variable for each entry in the configuration (i.e., for each register, for the memory value, for the memory address, etc.). The variables in the third set are auxiliary variables; their purpose is explained later.

Notation:

- Variables representing the current configuration – $V_{\text{curr}} = \{X_1, ..., X_N\}$.
- Variables representing the next configuration – $V_{\text{next}} = \{X'_1, X'_2, ..., X'_N\}$.
- Auxiliary variables – $V_{\text{aux}} = \{Y_{\text{aux}1}, Y_{\text{aux}2}, ..., Y_{\text{aux}l}\}$.
Any relation can be defined over the first two sets of variables, however, the degree of the polynomials needed to define some relations may be very large; It is sometimes beneficial, for both the prover’s and verifier’s practical efficiency, to add auxiliary variables, which are not part of \( V_{\text{curr}} \) and \( V_{\text{next}} \), in order to reduce the degree of the polynomials.

Henceforth, in order to distinguish in writing between variables and values, upper case letters denote variables while lower case denote values. Bold values \( x, y, \ldots \) are the values in the combinatorial world (sometime referred to as combinatorial values) and \( x, y, \ldots \) are elements of the field \( F \). Similarly, \( X, Y, \ldots \) are used to define combinatorial variables and \( X, Y, \ldots \) define algebraic variables ranging over \( F \).

In the introduction we discussed three main contributions of this thesis. Here we focus on two of them: (i) decreasing the number of variables by pre-selection and (ii) modular algebraic construction. Though we believe that the former makes a more valuable contribution than the latter, we first discuss the modular algebraic construction because the terms and notions we use in this section will be used later in decreasing the number of variables by pre-selection section. The chapter is divided into three sections: (1) Algebraic Modular Construction (Section 4.1), in which we present the concepts and notions of defining a constraint system that is satisfied if and only if each two sequential configurations belong to the relation induced by the transition function \( \delta \), when \( \delta \) is given as a sequential program. We define the constraint system in a modular way. We first define a constraint system for an opcode and then expand it to an instruction and a program. (2) Selector Polynomial (Section 4.2), where we discuss methods to select a variable from numerous of options, according to the instruction’s arguments or opcode. One technique is to use multiplexers as was done in the state-of-the-art work [BSCGT13]. The second technique, which we use, is to use selector polynomials. Informally, these selectors are polynomials that are set to a non-zero value for specific values and to 0 for all other values. (3) Decreasing number of variables by pre-selection (Section 4.3), which is the main contribution of our work. In this section, we show a way to reduce the number of variables in the system significantly. We use the same variables in several constraint systems, and with the help of selector polynomials, set to zero constraints that we do not want to verify. Using this technique significantly improves both the verifier’s and prover’s execution times and efficiency, for myriad practical uses.

### 4.1 Modular Algebraic Construction

We now explain how to build an algebraic constraint system that is set to zero iff each two consecutive configurations belong the relation induced by the transition function (Definition 3.2.1). Recall our transition function is given as a sequential program consisting of instructions (Section 3.3). Henceforward, we can say that we define a constraint system that is set to zero, iff each pair of consecutive configurations belongs the relation induced by a corresponding instruction in the program. Informally, the
polynomials in this constraint system should be set to zero iff (1) the opcode is executed correctly over (2) the constants and the variables in $V_{curr}$ and $V_{next}$, which represent the instruction’s arguments, (3) variables that represent registers that are not modified by the instruction should hold the same value in both $V_{curr}$ and $V_{next}$ and (4) the PC is changed according to the opcode. For example, assume the existence of a BT-RAM machine with four registers $PC, R_1, R_2, R_3$ that executes the following instruction:

$$R_1 = R_2 + R_3.$$  

We want to write an algebraic constraint system that is satisfied only by an assignment that meets the following conditions:

- The variable in the $V_{next}$ that represents $R_1$ is assigned the algebraic value representing $R_2 + R_3$.

- The variables that represent $R_2$ in both $V_{curr}$ and $V_{next}$, are assigned the same value. The same attribute applies to the $R_3$ register.

- Suppose that the variable representing the PC in $V_{curr}$ is assigned an algebraic value that represents the integer $a$. Then the variable in $V_{next}$ that represents the PC is assigned the algebraic value of $a + 1$.

We define these algebraic constraint systems in a modular way. We start by defining a constraint system for an opcode (Definition 3.3.1) and later expand it to a constraint system for an instruction (Definition 3.3.2). Recall that in these definitions we are looking at opcodes and at instructions as relations, and we define constraint systems that are set to zero iff the relations hold. We call such a constraint system an *Algebraic Computational Integrity Problem* (ACIP). Formally,

**Definition 4.1.1 (ACIP).** Let $R$ be a relation such that $R \subseteq \mathbb{F}^k \times \mathbb{F}^{k'}$; we say $R$ has $k$ inputs and $k'$ outputs. An ACIP for $R$ is a set of polynomials $P_1, \ldots, P_m \in \mathbb{F}[X_1, \ldots, X_k, X'_1, \ldots, X'_{k'}, Y_1, \ldots, Y_{\ell}]$ such that $(x, x') \in R$ if and only if there exists $y \in \mathbb{F}^\ell$ such that

$$P_1(x, x', y) = \ldots = P_m(x, x', y) = 0.$$

Figure 4.1 is a schematic description of the stages in the transition of a sequential program from one configuration to another. There are three main components in this figure: $ALU$, instruction pre-processing and instruction post-processing. Given an instruction $I$, the instruction pre-processing component selects values from the configuration or constants according to $I$’s (input) arguments. These values are transferred to the $ALU$, and $I$’s opcode is executed on these values. The results of the computation are transferred to instruction post-processing. This component assigns the ALU’s results to the registers in the next configuration, according to $I$’s (output) arguments.
Figure 4.1: A schematic description of how a sequential program evolves from one configuration to the next.

Our construction of ACIPs is implemented bottom-up according to Figure 4.1. We first define \(k+l\) auxiliary variables, \(Y_{arg1}, \ldots, Y_{argk}, Y_{output1}', \ldots, Y_{outputl}'\). We denote the first \(k\) variables, \textit{ALU input variables}, and the last \(l\) variables as \textit{ALU output variables}. For each opcode in the machine language, we define an ACIP over these variables (and auxiliary variables). These ACIPs are set to zero iff an opcode is executed correctly.

Then, we expand the opcode’s ACIP to the instruction’s ACIP, by adding polynomials that capture the behavior of instruction pre-processing and instruction post-processing.

Recall that an opcode can be described as a relation (Definition 3.3.1). We say that a set of polynomials over \(F\) is an ACIP for an opcode if, for each \((k+l)\) tuple that belongs to the opcode’s relation, the algebraic translation of this tuple and an assignment over \(F\) to all the auxiliary variables sets all the polynomials to zero.

**Definition 4.1.2 (ACIP for an opcode).** A constraint system for a \((k,l)\)-opcode, \(R_{op}\), is a set of polynomials \(\overrightarrow{P} = \{P_1, P_2, \ldots, P_t\}\) where \(P_i \in F[X_1, \ldots, X_k, X'_1, \ldots, X'_l, Y_1, \ldots, Y_n]\) for each \(0 \leq i \leq t\) such that \(\forall (x_1, \ldots, x_k, x'_1, \ldots, x'_l) \in F^k \times F^l:\)

\[
(x_1, \ldots, x_k, x'_1, \ldots, x'_l) \in R_{op} \iff \exists (y_1, \ldots, y_n) \in F^n \ \text{s.t.} \ (x_1, \ldots, x_k, x'_1, \ldots, x'_l, y_1, \ldots, y_n) \ \text{is a solution of} \ \overrightarrow{P}
\]

**Definition 4.1.3 (ACIP for an instruction).** A constraint system for the \((k+1,l+1)\) instruction is a set of polynomials \(\overrightarrow{P} = \{P_1, P_2, \ldots, P_{t'}\}\) over the variables \(V_{curr} \cup V_{next} \cup V_{aux}\) such that each polynomial \(P_i\) for \(1 \leq i \leq t'\) belongs to one of the disjoint sets.

1. \(P_i\) in constraint system for the instruction’s \((k,l)\)-opcode

2. \(P_i\) is set to zero iff the variables \(Y_{arg1}, Y_{arg2}, \ldots Y_{argk}\) are assigned the relevant inputs in \(V_{curr}\) or constants, and \(Y_{output1}', \ldots, Y_{outputl}'\), are assigned outputs in \(V_{next}\), according to the instruction’s arguments.

3. \(P_i\) is set to zero iff the variables in \(V_{next}\), which are not the \(PC\) and do not represent instruction’s output arguments, are assigned with the same field element as the matching variables in \(V_{curr}\).
4. $P_1$ is satisfied iff the $PC$ register is modified correctly.

The instruction constraint system comprises (1) the opcode’s constraints, (2) constraints that are satisfied if the “right” field elements are assigned to the ALU input and output variables, (3) constraints that are set to zero iff variables, which are not modified during the instruction’s execution, are assigned with same values both in $V_{curr}$ and $V_{next}$, and (4) constraints that are set to zero if the program flow is executed correctly. Given a program $P$, we define the program’s $ACIP$ as the union of all instructions’ $ACIP$s that appear in the program. Formally,

**Definition 4.1.4** (ACIP for a program). Given a program $P = \{I_1, \ldots, I_l\}$, the program’s $ACIP$ is the union of the $ACIP$s for all instructions, $I_1, \ldots, I_l$.

### 4.2 Selector Polynomial

In the constraint systems that conform to *instruction pre-processing* and *instruction post-processing* (see Figure 4.1) we define polynomials that select a variable from numerous variables. For example, when the $i^{th}$ instruction is executed, in *instruction pre-processing*, we select from $V_{curr}$ the correct variables for the ALU’s inputs (as stated in the instruction). In the same manner, in *instruction post-processing*, we select the variables in $V_{next}$ that will be assigned the ALU’s outputs. One way to implement this selection is by using a multiplexer as defined in [BSCGT13]. In Definition C.22 they stated that a multiplexer that selects one output from $2^m$ options uses at least $(2^m + m)$ variables and has a degree of $m + 1$. We call this option *post selection* because the multiplexer should be presented with all the options before choosing the correct one. Before presenting the second option and providing formal definitions, we start with a discussion and example that demonstrates the notion of *pre-selection*.

At first glance, the construction of the second and third types of constraints in Definition 4.1.3 seems pretty straightforward. If register $i$ is the first input argument in the $k^{th}$ instruction, then the value assigned to $X_{arg_1}$ should be equal to the value assigned to $X_{R_i}$. In the same manner, if register $j$ does not participate in the instruction, then it’s value after the execution of the instruction should not be modified. A simple constraint system is:

\[ X_{arg_1} - X_{R_i} \]
\[ X_{R_j} - X_{R'_j} \]  

(4.1)

Without modification, these polynomials may not always be satisfied. Assume that in instruction $k' \neq k$, register $i' \neq i$ is the first argument in the opcode. Accordingly, in the constraint system, there would be two similar constraints,

\[ X_{arg_1} - X_{R_i} \]
\[ X_{arg_1} - X_{R_{i'}} \]

27
Most likely somewhere during the program’s execution, registers $R_i$ and $R'_i$ contain different values. Hence, $X_{R_i}$ and $X_{R'_i}$ would be assigned different field elements, which implies that at least one of the above constraints is not satisfied.

One way to solve this problem is by using pre-selection, multiplying the polynomials above (in Equation 4.1) by a selector polynomial defined over the PC variables, such that this polynomial is set to $\neq 0$ when the $k$th instruction is executed and to 0 otherwise.

One type of selector polynomials is a univariate selector polynomial that is defined over the PC variable. The univariate selector is of the form,

$$Selector_k(X_{PC}) = (X_{PC} - e_k + 1) \prod_{i=0, i \neq k}^{\ell-1} (X_{PC} - e_i),$$

where $\ell$ is the length of the program and $e_i$ and $e_k$ are the field elements that represent integers $i$ and $k$. One can notice that if $X_{PC}$ is assigned a value from $S = \{e_0, e_1, \ldots, e_{\ell-1}\}$, then the polynomial is set to zero for each element in $S \setminus \{e_k\}$, and set to $\neq 0$ for $e_k$. In order to verify that $X_{PC}$ is assigned a value from $S$, one can use type checking constraints (Section 5.1). The main disadvantage of univariate selector polynomials is the degree of the polynomials. If these selectors are used, then the degree of the polynomials is at least $\ell$.

The second type of selector polynomials are multivariate selector polynomials. In this technique, instead of having one variable that represents the PC in $V_{curr}$ and $V_{next}$, we have $\lceil \log(\ell) \rceil$ variables that represent the PC variables in each set, where $\ell$ is the length of the program. By using multivariate selector polynomials we significantly decrease the polynomials’ degree by slightly increasing number of variables that are used in the system. The key idea behind multivariate selector polynomials is analogous to univariate selectors, with small modifications. In multivariate selectors, there are $\lceil \log(\ell) \rceil$ PC variables that can be assigned $S = \{0, 1\}$. Further, instead of translating $k$ into its algebraic representation, we look at the binary representation of $k$, $(k_{\lfloor \log \ell \rfloor - 1} \ldots k_0)$ such that $k = \sum_{i=0}^{\lfloor \log \ell \rfloor - 1} 2^i \cdot k_i$, and translate each 0 and 1 into the algebraic element 0, 1 (recall each field must contain a zero and unit element).

$$Selector_k(X_{PC}^0, \ldots, X_{PC}^{\lceil \log \ell \rceil - 1}) = \prod_{i=0}^{\lceil \log \ell \rceil - 1} (X_{PC}^i + e_{k_i} - 1).$$

We illustrate the correctness of the selector by an example. Assume a program contains eight instructions and our goal is to validate the fifth instruction. In such case, there are three PC variables, $X_{PC}^0, X_{PC}^1, X_{PC}^2$, and the selector polynomial is

$$Selector_5(X_{PC}^0, X_{PC}^1, X_{PC}^2) = X_{PC}^0 \cdot (X_{PC}^1 - 1) \cdot X_{PC}^2$$

One can notice that when the PC’s variables are assigned the binary representation of 5 (101), the polynomial is set to $\neq 0$; in every other case, the polynomial is set to zero. For
each $0 \leq i \leq \ell - 1$, multiplying instruction $i$’s polynomials by $\text{Selector}(i)$ ensures that when instruction $i$’s polynomials are verified, all the other constraints are set to zero.

\textbf{Note.} In this thesis, we define a concrete construction of an \textit{ACIP} over fields of characteristic-2, namely, $\mathbb{F}_{2^m}$ (Chapter 5). In these fields, the multivariate selector polynomial can result in one of two values 0 or 1.

\section{4.3 Decreasing the Number of Variables by Pre-selection}

There are three main parameters we take into consideration when building the \textit{ACIP}: (1) the total number of variables in the system; (2) the total number of polynomials in the system; and (3) the degree of the polynomials. These parameters later on influence that proving and verification times, the length of the communication between the prover and the verifier, and the PCP proof’s length ([BBC+16]). Each parameter can be optimized at the expense of the other two. For example, assume we have a polynomial of degree $n$,

$$ p(X_1, X_2, \ldots, X_n) = X_1 \cdot X_2 \cdot \ldots \cdot X_n. $$

We can reduce that degree to be 2 by adding $n - 2$ auxiliary variables and by substituting this polynomial by $n - 1$ polynomials, as in the following:

\begin{align*}
  p_1(X_1, X_2, Y_1) &= Y_1 + X_1 \cdot X_2 \\
  p_2(X_3, Y_2, Y_1) &= Y_2 + Y_1 \cdot X_3 \\
  &\vdots \\
  p_{n-1}(X_n, Y_{n-2}) &= Y_{n-2} \cdot X_n
\end{align*}

An interesting line of investigation could be to find the optimal balance among the three parameters. We leave this question for future work. However, we found out that reducing the number of auxiliary variables in the system significantly, while increasing the degree slightly has an overall beneficial impact on the system.

In this section we describe the main contribution of this work, \textit{Decreasing number of variables by pre-selection}. Recall that in the introduction we discussed a natural way to implement the ALU component shown in Figure 4.1. In this implementation, we define an \textit{ACIP} for all the opcodes in the system. Each opcode’s \textit{ACIP} is defined over a different set of auxiliary variables and then using a multiplexer, the correct outputs are selected. Figure 1.1 illustrates this solution. If the natural implementation is applied, then number of auxiliary variables in the system is at least:

$$ \sum_{i=1}^{n_{\text{ops}}} (\text{op\_variables}(i)) + O(n_{\text{op}}), \quad (4.3) $$
when \( n_{\text{ops}} \) is the number of opcodes, and \( \text{op\_variables}(i) \) is the number of auxiliary variables each opcode uses. Assume that \( n_{\text{ops}} = 32 \) and the number of auxiliary variables each opcode uses is \( 2 \cdot W \) when \( W \) is the size in bits of a register. The total number of auxiliary variables in the system, consequently, is at least \( 64 \cdot W \). When the register size is 16 bits, this number equals 1024, and when register size is 32, the number of auxiliary variables in the system is at least 2048. When the average number of auxiliary variables an opcode uses is higher than \( 2 \cdot W \), as happens in practice (see Table 5), this number increases.

In this work, we show how to decrease number of auxiliary variables used by the ALU component (Figure 4.1) to

\[
\max_{i=1}^{n_{\text{ops}}}(\text{op\_variables}(i)),
\]

which clearly holds:

\[
\sum_{i=1}^{n_{\text{ops}}}(\text{op\_variables}(i)) \geq \max_{i=1}^{n_{\text{ops}}}(\text{op\_variables}(i)).
\]

The value in 4.4 can be further reduced to

\[
\max_{i=1}^{k}(\text{op\_variables}(i)),
\]

when \( k \) indicates the number of different opcodes in a program, \( P \). When a program uses only a subset of opcodes, the number of variables in the natural way of implementation can be decreased as well. It will still, however, a summation of the number of auxiliary variables each opcode in the program uses.

**Theorem 4.1** (The program’s ACIP measurements). *Given a program \( P = \{I_1, I_2, \ldots, I_\ell\} \), we denote by \( \text{op}_1, \ldots, \text{op}_k \) the different opcodes in the program; clearly \( k \leq \ell \). The number of auxiliary variables in the program’s ACIP is

\[
\max_{i=1}^{k}(\text{op\_variables}(i)) + O(1),
\]

and the maximum degree is

\[
\max_{i} \{\deg(p_i) | p_i \in \text{ACIP}\} + \lceil \log(\ell) \rceil,
\]

where \( \text{op\_variables}(i) \) is the number of auxiliary variables used in \( \text{op}_i \)’s ACIP, and \( \deg(p_i) \) is a degree of a polynomial.

**Proof.** We define a global set of auxiliary variables of size \( \max_{i=1}^{k}(\text{op\_variables}(i)) \). For each opcode that appears in the program, we define an ACIP over the global set of variables. We multiply each opcode’s ACIP by an appropriate multivariate selector polynomial (Equation 4.2). In other words, if \( \text{op}_i \) appears in instructions \( I_{j_1}, \ldots, I_{j_n} \),
(n ≤ ℓ), we multiply all the polynomials in op_i’s ACIP by

\[ \sum_{i=1}^{n} \text{Selector}_{j_i} \left( X_{PC}^{0}, \ldots, X_{PC}^{[\log \ell] - 1} \right). \]

The multiplication of I_i’s constraints by a selector polynomial ensures that when I_i is executed, only its polynomials are verified and the other polynomials are automatically set to 0. The multiplication by a multivariate selector polynomial increases the degree of the polynomials slightly by \([\log(\ell)]\), which is number of bits needed to represent each instruction in the program.

The O(1) that is used in the number of auxiliary variables formula, represents auxiliary variables that are used for general purposes, such as representing the ALU’s input and output variables.

As mentioned in the beginning of section 4.3, it is possible to improve one parameter at the expense of the others. In our work, we significantly improve the number of variables in the ACIP by increasing the maximum degree of a polynomial in the system slightly. This method is in contrast to the natural way, where number of variables is high and the max degree can be described as

\[ \max \{ \deg(p_i), \deg(MUX) \}, \]

where p_i is a polynomial in the ACIP and MUX represents the multiplexer with the highest number of input. As stated in [BSCGT13], a multiplexer that chooses one output out of \(2^m\) would be represented by a polynomial of degree \(m + 1\). We found, that in practice, for the programs we ran, our method has an overall advantage over the natural way.

Assume a program that consists of two instructions: the first one executes a bitwise \textit{AND} and the second one executes a bitwise \textit{OR}. In our implementation, each opcode uses \(2 \cdot W + 1\) auxiliary variables (Table 5), where \(W\) is the register size. Because of \textit{reuse of variables}, there are only \(2 \cdot W + 1\) ALU auxiliary variables in the system, instead of \(4 \cdot W + 2\). Moreover, when the second instruction is verified, \textit{AND}’s constraints are automatically set to zero, because they are multiplied by a selector polynomial that equals 1 only when the PC points to the first instruction; in all other cases, it equals 0.

A more practical example is the exhaustive subset-sum program defined in [BBC+16, Code 2]. The subset-sum is an important problem in complexity theory, which gets two inputs, a multi-set of integers, \(A\), and a number \(n\), and returns true if there exists a subset of integers in the multi-set whose sum is \(n\). The exhaustive subset-sum presented in the paper tries all the possible subsets, \(2^{[A]}\) options, and returns whether there exists a subset whose sum is \(n\). The exhaustive subset-sum program presented in this paper, is written in TinyRAM [BCG+13, BSCG+13]. It is executed on a 16-bit machine \((W = 16)\), which has 16 registers. The program is 22 lines long; therefore, the PC is
represented by five variables in $V_{curr}$ and $V_{next}$. Both $V_{next}$ and $V_{curr}$ are represented by 22 variables. The number of variables used in order to define an ACIP for the program is 114; thus, number of auxiliary variables in the system is 70. This value makes sense because an ADD opcode, which appears in the eleventh instruction, uses $4 \cdot W - 1 = 63$ variables. Other auxiliary variables are used for the system’s general purpose, such as the ALU input and output variables. There are only 85 polynomials in the system, and the highest degree is 12. If the ACIP was defined in the natural way, then number of auxiliary variables would be at least $150, \times 2.1$ more than in our implementation. This high number of auxiliary variables is a result of the use of opcodes, AND, ADD, SHR that on average require at least $3 \cdot W$ auxiliary variables (see Table 5). The degree would be slightly decreased to $12 - 5 = 7$, and the number of polynomials would be at least the order of the variables. In Section 5.3, one can find next to each opcode, the number of polynomials, the degree, and the number of auxiliary variables each opcode’s ACIP requires. By using our method, one can see that the number of polynomials in the program’s ACIP is reduced as well, which leads us to our next point.

Another advantage of the reuse of the variables is that it enabled us to reduce the number of polynomials in the system. We take advantage of the fact that in many opcodes, the auxiliary variables have the same “role”. In other words, there are several variables that can be assigned with the same values. For example, in bitwise-OR and bitwise-AND, we use $2 \cdot W + 1$ variables in order to define an ACIP (see Table 5), and $2 \cdot W$ of these variables are used to present the numbers as a binary vectors. In such cases, it is necessary to validate that each variable is assigned only 0 or 1. This can be done by using type checking polynomials (Section 5.1). When variable reuse is applied, it is sufficient to check that these variables are assigned 0 or 1 only for the global set of variables, and not for every allocated copy.
Chapter 5

ACIP construction over $\mathbb{F}_{2^m}$

Recall in the introduction we stated that a main contribution of this thesis is that we succeeded in efficiently encoding the transition function as a set of polynomials over small characteristic fields. In this chapter we focus on defining the ACIPs over fields of small characteristics, especially on $\mathbb{F}_{2^m}$. We are one of the first works that use small characteristic fields in order to encode a computation. Many previous works such as [BCG+13, SMBW12, SVP+12] focused on encoding a computation over large prime fields. Generally speaking, the operations in large prime fields resemble addition and multiplication of integers (excluding overflow and borrow); therefore, defining ACIPs over such fields can be considered more natural for many opcodes than over $\mathbb{F}_{2^m}$.

In Chapter 4 we discussed the importance of minimizing the number of variables used in the program's ACIP. The main concern of defining ACIPs over $\mathbb{F}_{2^m}$ is that some opcodes would require $\Omega(W^2)$ auxiliary variables, when $W$ represents a register size. If these opcodes are used by some program, the number of variables in the system can be very high. An excellent example to demonstrate this concern is integer multiplication. When defining an ACIP for multiplication the objective is to define polynomials that are set to zero iff the multiplication of $W$-bits integers is executed correctly. The most intuitive way to define this ACIP over $\mathbb{F}_{2^m}$, is to represent the integers as $\{0, 1\}$-vectors over $\mathbb{F}_{2^m}$ and then mimic binary multiplication behavior. The main disadvantage of this technique is that it requires $W$ additions of size $W$. In other words, there would be at least $W^2$ auxiliary variables in such a constraint system.

A key contribution of this work is that we successfully defined an ACIP for numerous widely-used opcodes, such that each opcode uses only $O(W)$ auxiliary variables. Recall in Section 4.3 that we showed that the number of auxiliary variables in the program ACIP is roughly the maximum number of auxiliary variables an opcode in the program uses (Theorem 4.1). Therefore, in our implementation, the total number of auxiliary variables in a program would be linear in $W$. We chose to define concrete ACIPs for opcodes based on TinyRAM's assembly language [BSCG+13, BCG+13]. TinyRAM is a random access machine (RAM) whose assembly language is, on the one hand, concise and contains only 27 opcodes, yet on the other hand, these opcodes are very general.
and used in many widely-used assembly languages. The arithmetic in this machine is performed over the integers. Floating-point arithmetic is not defined by TinyRAM’s assembly; however, it can be implemented with the given opcodes. In Table 5 we summarize the number of auxiliary variables, the maximum degree and the number of polynomials for our ACIPs, which the opcodes require in our implementation:

<table>
<thead>
<tr>
<th>opcode</th>
<th>#auxiliary variables</th>
<th>max degree</th>
<th>#polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>not</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>and</td>
<td>2 \cdot W + 1</td>
<td>2</td>
<td>2 \cdot W + 5</td>
</tr>
<tr>
<td>or</td>
<td>2 \cdot W + 1</td>
<td>2</td>
<td>2 \cdot W + 5</td>
</tr>
<tr>
<td>add</td>
<td>4 \cdot W - 1</td>
<td>2</td>
<td>4 \cdot W + 3</td>
</tr>
<tr>
<td>sub</td>
<td>4 \cdot W - 1</td>
<td>2</td>
<td>4 \cdot W + 3</td>
</tr>
<tr>
<td>mul</td>
<td>8 \cdot W</td>
<td>4</td>
<td>8 \cdot W + 6</td>
</tr>
<tr>
<td>umull</td>
<td>8 \cdot W</td>
<td>4</td>
<td>8 \cdot W + 6</td>
</tr>
<tr>
<td>smulh</td>
<td>8 \cdot W</td>
<td>4</td>
<td>8 \cdot W + 6</td>
</tr>
<tr>
<td>udiv</td>
<td>9 \cdot W + 1</td>
<td>4</td>
<td>9W + 7</td>
</tr>
<tr>
<td>umodh</td>
<td>9 \cdot W + 1</td>
<td>4</td>
<td>9W + 7</td>
</tr>
<tr>
<td>cmpe</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>cmpa</td>
<td>3 \cdot W</td>
<td>4</td>
<td>3 \cdot W + 3</td>
</tr>
<tr>
<td>cmpg</td>
<td>3 \cdot W</td>
<td>4</td>
<td>3 \cdot W + 3</td>
</tr>
<tr>
<td>cmpae</td>
<td>3 \cdot W</td>
<td>4</td>
<td>3 \cdot W + 3</td>
</tr>
<tr>
<td>cmpge</td>
<td>3 \cdot W</td>
<td>4</td>
<td>3 \cdot W + 3</td>
</tr>
<tr>
<td>shl</td>
<td>3 \cdot W + \lceil \log W \rceil + 3</td>
<td>2</td>
<td>3 \cdot W + \lceil \log W \rceil + 7</td>
</tr>
<tr>
<td>shr</td>
<td>3 \cdot W + \lceil \log W \rceil + 3</td>
<td>2</td>
<td>3 \cdot W + \lceil \log W \rceil + 7</td>
</tr>
</tbody>
</table>

Table 5.1: Number of auxiliary variables, maximum degree and number of polynomials for a subset of opcodes

In order to present a concrete constructions of ACIPs over fields of small characteristic, we need to make several assumptions. These assumptions are changeable; however one may need to adjust some of the constraints.

- The algebraic field \( \mathbb{F} \) is a characteristic-2 extension field, \( \mathbb{F}_{2^m} \).
- \( W \) is a register’s length and \( m > 2 \cdot W \).
- All BT-RAM’s alphabets \( (\Sigma_i, \Sigma_r, \Gamma_a, \Gamma_v) \) equal \( \Sigma_a = \{0, 1\}^W \). In order to support special characters that appear in these alphabets, we define an augmented configuration, which indicates whether the special character is used in the original configuration. For example, instead of using the \( \perp \) sign, one can define a new field isMemOp which is set if and only if memory address \( C|\Gamma_a \neq \perp \).
- In \( S_{\text{memory sorted}} \), a memory trace sorted first by memory address and then by time, the configurations that do not access the memory module \( C|\Gamma_a = \perp \) appear at the end.
Signed integers are represented with two’s complement technique (see Section 5.2).

Chapter 5 is divided into four sections. In section 5.1 we explain how to write polynomials that are satisfied iff a variable is assigned specific values. In Section 5.2, *Integer Representation over \( \mathbb{F}_{2^m} \)*, we focus on methods that can be applied in order to translate integers into field elements. In Section 5.3, *ALU Opcodes*, we present ACIPs for different widely-used opcodes. We defined ACIPs for opcodes based on TinyRAM assembly, and our main achievement, is that each ACIP uses only a linear in \( W \) number of variables. In Section 5.4, *Memory validity*, we show how to define a memory ACIP over \( \mathbb{F}_{2^m} \) such that, given a memory-sorted trace \( S_{\text{memory \_sorted}} \), these polynomials are set to zero if \( S_{\text{memory \_sorted}} \) is sorted first by address and then by memory as defined in Definition 3.2.3 and is memory valid as defined in Definition 3.2.2.

### 5.1 Type Checking

In many cases we want to check that a variable is assigned a value from a specific set of values, \( S \subseteq \mathbb{F} \). An example is the flag register, which in many systems may hold only a binary value, \{0, 1\}. Consequently, we should define a constraint that is satisfied if and only if the variable is assigned a value from \( S \). We may say that the constraint forces the variable to be assigned a specific value.

**Lemma 5.1.1.** Given a variable \( X \) and set \( S \subseteq \mathbb{F} \), the polynomial:

\[
C_{\text{type}}(X) = \prod_{a \in S} (X - a),
\]

is set to zero if and only if \( x \in S \).

**Proof.** One can see that if \( x = a \) such that \( a \in S \), then the polynomial is set to zero. Since (5.1) has a degree \( |S| \), it has at most \( |S| \) roots. Meaning that if all the roots are in \( S \), then if \( x \in \mathbb{F} \setminus S \), then 5.1 is not satisfied.

### 5.2 Integer Representation over \( \mathbb{F}_{2^m} \)

We now present several ways to translate an integer into a field element. We first start with a discussion about integer representation in the combinatorial world, and then focus on the representation of integers as element over \( \mathbb{F}_{2^m} \).

Every positive integer, \( a \), has a unique representation of the form \( a = \sum_{i=0}^{n} a_i \cdot 2^i \), where \( a_i \in \{0, 1\} \). There are several ways to represent negative numbers. The literature focuses on three main techniques: *sign and magnitude*, *one’s complement* and *two’s complement*. Currently, the most common representation is *two’s complement*, because each number has a unique representation (as opposed to 0 in the other two techniques)
and it allows the execution of signed arithmetic operations in the same way as in unsigned numbers. In this technique, signed integers are represented by

\[ a = -a_n \cdot 2^n + \sum_{i=0}^{n-1} a_i \cdot 2^i. \]

The leading bit represents the sign. If it is equal to 1, then the number is negative; otherwise, it is positive or zero. In order to represent a negative number, one should look at the unsigned representation, swap all the bits and add 1. For example, \(-1\) equals

\[ 001 \xrightarrow{\text{swap}} 110 \xrightarrow{+1} 111, \]

when represented with three bits. One can verify that this sequence equals \(-1\) by placing the bits into the above formula, to get:

\[ -1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -4 + 2 + 1 = -1, \]

Clearly, there are many possibilities to represent an integer as a field element. We are going to show three different ways that we think are more natural than the others and will be used later to define the program’s \textit{ACIP}.

- **Additive** Let \(\beta_0, \ldots, \beta_{m-1}\) be a standard basis for \(F_{2^m}\) over \(F_2\). This means every element of \(F_{2^m}\) can be uniquely represented as \(\sum_i x_i \beta_i\) where \(x_i \in F_2\). The \textit{additive} representation of \(n\) (under the aforementioned basis) is simply \(\sum_{i=0}^{m-1} n_i \beta_i\) where \(n_i \in \{0, 1\}\) is the binary representation of \(n\).

- **Additive unpacked** Recalling \(F_2 \subset F_{2^m}\), we may use the binary expansion of \(n\) to represent it as a sequence of bits, namely, the sequence \((n_0, \ldots, n_{m-1}) \in (F_2)^m\). This is called the \textit{unpacked additive} representation of \(n\).

- **Multiplicative** Let \(g\) be a generator of the multiplicative group \(F_{2^m}^*\); recall this is a cyclic group of order \(2^m - 1\). The \textit{multiplicative} representation of \(n\) is \(g^n\); notice the mapping \(n \mapsto g^n\) is injective on \(\{0, 1, \ldots, 2^m - 2\}\) and \(k < m\) so it is also injective on \(\{0, 1, \ldots, 2^k - 1\}\).

### 5.2.1 Constraint System for Transition between Representation Methods

The representation methods described above may be used not only to translate between an integer and a field element, but also to convert between different representations of a field element, in order to write efficient constraint systems. For example, assume we get two integers in the \textit{additive unpacked} representation and we want to write a constraint system for \textit{XOR} opcode. The straightforward solution is to define a \(W\) auxiliary
variables, such that each auxiliary variable equals the XOR result on the matching inputs’ variables. A more sophisticated and efficient solution would be to convert the additive unpacked representations to additive representations and then use addition over $\mathbb{F}_{2^m}$. Converting between these representations can help us define constraint systems with fewer polynomials, fewer auxiliary variables or polynomials of lower degree. We focus on writing a constraint system that captures the translation between additive unpacked and additive representations (and visa versa), and the translation between additive unpacked and multiplicative representations (and visa versa).

Additive Unpacked vs. Additive Representations

**Definition 5.2.1** (Additive representation). Let $B = \{\beta_0, \ldots, \beta_{m-1}\}$ be a standard basis for $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. For $x, y_0, \ldots, y_{m-1} \in \mathbb{F}_2^{m+1}$, we say $\vec{y}$ is the unpacking of $x$, and $x$ is the packing of $\vec{y}$ under basis $B$ if

1. $y_i \in \mathbb{F}_2$ for $i = 0, \ldots, m-1$
2. $x = \sum_{i=0}^{m-1} y_i \beta_i$.

**Lemma 5.2.2** (Constraint system for additive packing). Let $Y_0, Y_1, \ldots, Y_{m-1}$ be the unpacked variables and $X$ be the packed variable. The constraint system matching additive packing (5.2.1) is, therefore,

$$C_{\text{additive packing}}(Y_0, Y_1, \ldots Y_{m-1}, X) = \left\{ X + \sum_{i=0}^{i=m-1} Y_i \cdot \beta_i \right\} \cup \{Y_i \cdot (Y_i + 1) \mid 0 \leq i \leq m - 1\}$$

From 5.2.1, it is clear that the constraint system is satisfied if and only if $x$ is the packed representation of $\vec{y}$. One can notice that the constraint system for converting from packing to unpacking representations and from unpacking to packing is the same.

$$C_{\text{additive unpacking}}(X, Y_0, Y_1, \ldots Y_{m-1}) = C_{\text{additive packing}}(Y_0, Y_1, \ldots Y_{m-1}, X)$$

The second type of polynomials (in Lemma 5.2.2) is type checking constraints (Section 5.1). Although it is necessary to use type checking constraints to enforce specific values on a variable for soundness reasons, there is no need to apply these constraints more than once on the same variable. If a variable appears in several constraint systems, one should avoid adding redundant type checking constraints. For example, in additive packing if the type of $Y_i$ for $0 \leq i \leq m - 1$ has already been checked, we can omit these constraints in 5.2.2.
Additive Unpacked vs. Multiplicative Representations

Definition 5.2.3 (Multiplicative representation for unsigned integers). Let \( g \) be the generator of \( \mathbb{F}_{2^m}^* \). For \( x \in \mathbb{F}_{2^m}^* \) and \( y_0, \ldots, y_{m-1} \in \mathbb{F}_{2^m} \), we say \( \vec{y} \) is the multiplicative unpacking of \( x \), and \( x \) is the multiplicative packing of \( \vec{y} \) if

1. \( y_i \in \mathbb{F}_2 \) for \( i = 0, \ldots, m-1 \)
2. \( y = \sum_{i=0}^{m-1} 2^i \cdot y_i \)
3. \( x = g^y \).

Lemma 5.2.4 (Constraint system for multiplicative packing for unsigned integers). Let \( Y_0, Y_1, \ldots, Y_{m-1} \) be the additive unpacked variables, \( X \) be the multiplicative packed variable, and \( Z_0, \ldots, Z_{m-1} \) the auxiliary variable. The constraint system matching multiplicative packing (5.2.3) is, therefore,

\[
C_{\text{multiplicative packing}}(\vec{Y}, X, \vec{Z}) = \{Y_i \cdot (1 + Y_i)|0 \leq i \leq m - 1\} \cup \{Z_0 + Y_0 \cdot g + 1 + Y_0\} \cup \{Z_i + Z_{i-1} \cdot (Y_i \cdot g^{2^i} + 1 + Y_i)|1 \leq i \leq m - 1\} \cup \{Z_{m-1} + X\}
\]

Proof. We want to show that (5.2.4) is the constraint system for (5.2.3). The first set of constraints is type checking constraints (Section 5.1) that verify that \( y_i \in \mathbb{F}_2 \) for each \( 0 \leq i \leq m - 1 \). We prove that the polynomials are satisfied iff \( Z_i \) is assigned \( z_i = g^{\sum_{j=0}^{i} 2^j y_j} \) by induction on \( i \).

base case \( i=0 \): \( y_0 \in \{0, 1\} \) thus,

\[
z_0 = g^{\sum_{j=0}^{0} 2^j y_j} = g^{y_0} = \begin{cases} 1 & y_0 = 0 \\ g & y_0 = 1 \end{cases} \tag{5.2}
\]

We want to show that the constraint \( Z_0 + Y_0 \cdot g + 1 + Y_0 \) is set to zero iff \( Z_0 \) is to be assigned the correct value according to 5.2. If \( Y_0 \) is assigned zero, the polynomial is \( Z_0 + 1 \), and is satisfied if and only if \( z_0 = 1 \). If \( Y_0 \) is assigned 1, the polynomial is \( Z_0 + g + 1 = Z_0 + g \), and is satisfied if and only \( z_0 = g \).

Let’s assume the correctness of the claim for \( i-1 \) and prove for \( i \) such that \( i \geq 1 \).

\[
z_i = g^{\sum_{j=0}^{i} 2^j y_j} = g^{\sum_{j=0}^{i-1} 2^j y_j} \cdot g^{2^i y_i} = \begin{cases} z_{i-1} & y_i = 0 \\ z_{i-1} \cdot g^{2^i} & y_i = 1 \end{cases} \tag{5.3}
\]

The constraint \( Z_i + Z_{i-1} \cdot (Y_i \cdot g^{2^i} + 1 + Y_i) \) is set to zero iff \( Z_i \) is assigned according to Equation 5.3. If \( Y_i \) is assigned 0, the polynomial equals \( Z_i + Z_{i-1} \), which is satisfied if
and only if $Z_i$ is assigned $z_{i-1}$. If $Y_i$ is assigned 1, the polynomial equals $Z_i + Z_{i-1} \cdot g^{-1}$, which is clearly satisfied if only if $Z_i$ is assigned $z_{i-1} \cdot g^{-1}$.

From this induction, it is clear that $z_{m-1} = g \sum_{i=0}^{m-1} 2^i y_i$. The last constraint verifies that $X$ is indeed the multiplicative packing of $y$. One can notice that the last constraint is redundant, and we can eliminate it by changing the constraint system to (when $m \geq 3$):

$$C_{\text{multiplicative packing}}(\vec{Y}, X, \vec{Z}) =$$

$$\{ Y_i \cdot (1 + Y_i) | 0 \leq i \leq m - 1 \} \cup$$

$$\{ Z_0 + Y_0 \cdot g + 1 + Y_0 \} \cup$$

$$\{ Z_i + Z_{i-1} \cdot (Y_i \cdot g^{2^i} + 1 + Y_i) | 1 \leq i \leq m - 2 \} \cup$$

$$\{ X + Z_{m-2} \cdot (Y_{m-1} \cdot g^{2^m-1} + 1 + Y_{m-1}) \}$$

If the type of the variables $Y_i$ for $0 \leq i \leq m - 1$ has been enforced, the type checking constraints are redundant in this constraint system. The multiplicative unpacking constraint system is the same as multiplicative packing.

$$C_{\text{multiplicative unpacking}}(X, \vec{Y}, \vec{Z}) = C_{\text{multiplicative packing}}(\vec{Y}, X, \vec{Z}) \quad (5.4)$$

**Signed Representation**  If $y$ is a negative number then in two’s complement it is defined as

$$y = -2^{m-1} + \sum_{i=0}^{m-2} 2^i \cdot y_i.$$  

Our objective is to transform it into $g^y = \frac{1}{g^y}$. Therefore, when a signed transformation is required, we substitute the last constraint

$$\{ X + Z_{m-2} \cdot (Y_{m-1} \cdot g^{2^m-1} + 1 + Y_{m-1}) \}$$

by

$$\{ X + Z_{m-2} \cdot (Y_{m-1} \cdot g^{-2^m-1} + 1 + Y_{m-1}) \}.$$  

### 5.3 ALU Opcodes

In this section we present widely-used opcodes based on TinyRAM [BSCG+13, BCG+13] assembly language. TinyRAM is a random access machine whose assembly language is, on the one hand, concise and contains only 27 opcodes, yet, on the other hand, is expressive and consists of widely-used opcodes. The arithmetic in this machine is performed over the integers. Floating-point arithmetic is not defined by TinyRAM’s
assembly, but, it can be easily implemented with the given TinyRAM opcodes. We focus on writing ACIPs for opcodes that are “challenging”. We omit opcodes such as `mov`, whose constraint systems are trivial. The opcodes we present are from a spectrum of operations, and can be divided logically into five disjoint sets: binary, integer, comparison, shift, jump.

The majority of the presented opcodes in the section are denoted by four-arity relation. They have 1 or 2 input values \( (k \leq 2) \) and 1 or 2 output values \( (l \leq 2) \). We indicate the variables that represent the ALU input variables as \( X_{arg1} \) and \( X_{arg2} \), and the variables that represent the ALU output variables as \( X'_{output} \) and \( X'_{flag} \). Flag is a status register, which in our system, can hold one of two values \{0, 1\}. Many known architectures have several flag registers. These registers help the user to check certain conditions while the code is executed, i.e., a carry flag indicates if an arithmetic carry or borrow happened, an overflow flag indicates if an arithmetic overflow has occurred. TinyRAM has only one flag, and most opcodes can modify it.

In order to avoid repetitions, we define the ACIPs in a modular way. We start by presenting ACIPs for auxiliary opcodes. These constraint systems are used in several opcodes. For example, in Section 5.3.1, we define an ACIP for the comparison opcode, which returns one of three values that indicate whether the first integer is greater, smaller or equal to the second integer. In Section 5.3.4 we use this ACIP in order to define the different TinyRAM comparison opcodes. We elaborate on the ACIP construction of each opcode in the disjoint set of operations. For each ACIP, we state the number of auxiliary variables it requires, the maximum degree of a polynomial, and the number of polynomials in the ACIP. Also, we show that each constraint system is, in fact, an ACIP for an opcode based on Definition 4.1.2.

### 5.3.1 Auxiliary Opcodes

**Indicator**

#auxiliary variables: 1, max degree: 2, #polynomials: 2

This is a relation over two values. If the first value equals \( c \in \Sigma_b \), the second value is 1; otherwise the second value is 0. Formally,

\[
\mathcal{R}_{\text{Indicator}_c}(x, x') = \{(x, 1) | x = c\} \cup \{(x, 0) | x \neq c\}. \tag{5.5}
\]

We can define the next ACIP:

\[
\mathcal{C}_{\text{Indicator}_c}(X, X', Y) = \{(X + e_c) \cdot X'\} \cup \{(X + e_c) \cdot Y + 1 + X'\}, \tag{5.6}
\]

where \( e_c \) is the element in \( \mathbb{F} \) representing \( c \).

**Lemma 5.3.1.** \( (x, x') \in \mathcal{R}_{\text{Indicator}_c} \iff \exists y \in \mathbb{F} \text{ s.t. } \mathcal{C}_{\text{Indicator}_c}(x, x', y) \text{ is satisfied.} \)

**Proof.** \( \Leftarrow \) The first constraint is satisfied iff \( x = e_c \) or \( x' = 0 \).
• $X$ is assigned $e_c$. Then the second constraint equals $X' + 1$. This constraint is satisfied if $x' = 1$.

• $X'$ is assigned with 0. Then the second constraint is $(X + e_c) \cdot Y + 1$. Clearly, $x \neq e_c$ (if $x = e_c$, the second polynomial is not satisfied), and therefore, $x + e_c \neq 0$. Recall that each field element ($\neq 0$) has an inverse; thus if $Y$ is assigned $(x + e_c)^{-1}$, the constraint is set to zero.

In both cases, it is clear that $(x, x') \in R_{\text{Indicator}}$.

⇒ Obviously, if $(x = e_c \land x' = 1)$ or $(x \in \mathbb{F} \setminus \{e_c\} \land x' = 0 \land y = (x + e_c)^{-1})$ the ACIP is satisfied.

**compare**

#auxiliary variables: $W - 1$, max degree: 4, #polynomials: $3W$

Compare compares two binary representations of integers. It returns 1 if the first integer is greater than the second, $g$ if the integers are equal and 0 otherwise. One can define the comparison relation as $(2 \cdot W, 1) – \text{compare opcode}$

$$
R_{\text{compare}}(\bar{x}_{arg1}, \bar{x}_{arg2}, x'_\text{output}) =
\begin{cases}
\{ (\bar{x}_{arg1}, \bar{x}_{arg2}, 1) | x_{arg1} > x_{arg2} \} \
\{ (\bar{x}_{arg1}, \bar{x}_{arg2}, 2) | x_{arg1} = x_{arg2} \} \
\{ (\bar{x}_{arg1}, \bar{x}_{arg2}, 0) | x_{arg1} < x_{arg2} \}
\end{cases}
\quad (5.7)
$$

such that $\bar{x}_{argi}$ is the binary representation of $x_{argi}$ for $i \in \{1, 2\}$.

One way to compare two binary integer representations is by scanning the bits from the most significant bit (MSB) to the least significant bit (LSB). The first pair of bits such that $x_{arg1}^i \neq x_{arg2}^i$ determines the result of the comparison. We define an ACIP that is based on this technique of comparison. This ACIP uses $W$ auxiliary variables, $Y^1_{aux} = \{ Y_{aux}^{W-1}, \ldots, Y_{aux}^0 \}$, which can be assigned one of three values, $\{0, 1, g\}$. If $Y^{W-i}$ is assigned either 1 or 0, it means that the comparison result was determined while scanning the $i$ most significant bits. An assignment of 1 denotes that the first integer is bigger than the second, and an assignment of 0 denotes the opposite. Clearly, in such cases, the comparison between the $W - i$ least significant bits does not affect the result of the comparison. If $Y^{W-i}$ is assigned $g$, the $i$ most significant bits are identical in both integers. The value of $y_{aux}^i$ can be defined based on the $i^{th}$ bits of the input and $y_{aux}^{i+1}$, namely,

$$
y_{aux}^i =
\begin{cases}
1 & (i = W - 1 \land x_{arg1}^i = 1 \land x_{arg2}^i = 0) \\
(\neg i < W - 1 \land (y_{aux}^{i+1} = 1 \lor (y_{aux}^{i+1} = g \land x_{arg1}^i = 1 \land x_{arg2}^i = 0))) \\
0 & (i = W - 1 \land x_{arg1}^i = 0 \land x_{arg2}^i = 1) \\
(\neg i < W - 1 \land (y_{aux}^{i+1} = 0 \lor (y_{aux}^{i+1} = g \land x_{arg1}^i = 0 \land x_{arg2}^i = 1))) \\
g & (i = W - 1 \lor (i < W - 1 \land y_{aux}^{i+1} = g)) \land x_{arg1}^i = x_{arg2}^i
\end{cases}
\quad (5.8)
$$
We start by defining an ACIP for unsigned integers, and later we expand this ACIP to signed integer representations as well. Based on Equation 5.8, we define the following ACIP:

$$C_{\text{compare}}(\vec{X}_{\text{arg}1}, \vec{X}_{\text{arg}2}, X'_{\text{output}}, Y_{\text{aux}}) =$$

$$\{ Y_{\text{aux}}^i \cdot (Y_{\text{aux}}^i + 1) \cdot (Y_{\text{aux}}^i + g) \mid 0 \leq i < W \} \cup$$

$$\{ X_{\text{arg}1}^i \cdot (X_{\text{arg}1}^i + 1) \mid 0 \leq i < W \} \cup$$

$$\{ X_{\text{arg}2}^i \cdot (X_{\text{arg}2}^i + 1) \mid 0 \leq i < W \} \cup$$

$$\{ p_{\text{compare}}^i \mid 0 \leq i < W \} \cup$$

$$\{ X'_{\text{output}} + Y_{\text{aux}}^0 \}$$

where:

$$p_{\text{compare}}^{W-1} = Y_{\text{aux}}^W + g \cdot (X_{\text{arg}1}^{W-1} + X_{\text{arg}2}^{W-1} + 1) + (X_{\text{arg}1}^{W-1} \cdot (X_{\text{arg}2}^{W-1} + 1))$$

$$p_{\text{compare}}^i = Y_{\text{aux}}^i +$$

$$+ Y_{\text{aux}}^{i+1} \cdot (1 + Y_{\text{aux}}^{i+1}) \cdot (g \cdot (1 + g))^{-1} \cdot [g \cdot (X_{\text{arg}1}^i + X_{\text{arg}2}^i + 1) + X_{\text{arg}1}^i \cdot (X_{\text{arg}2}^i + 1)]$$

$$+ Y_{\text{aux}}^{i+1} \cdot (g + Y_{\text{aux}}^{i+1}) \cdot (1 + g)^{-1}$$

**Lemma 5.3.2.** \((\vec{X}_{\text{arg}1}, \vec{X}_{\text{arg}2}, X'_{\text{output}}) \in R_{\text{compare}} \iff \exists Y_{\text{aux}}^W \text{ s.t. } C_{\text{compare}}(\vec{X}_{\text{arg}1}, \vec{X}_{\text{arg}2}, X'_{\text{output}}, Y_{\text{aux}}) \text{ is satisfied}$$

**Proof.** We focus on \(\Leftarrow\) side of the proof because \(\Rightarrow\) is trivial. We leave it to the reader to check that the assignment matches Relation 5.7 and Equation 5.8 sets the constraints to zero. Also, we do not elaborate on the first three sets of polynomials. These are type checking constraints that verify that \(X_{\text{arg}1}^i\) and \(X_{\text{arg}2}^i\) are assigned \(\{0, 1\}\) and \(Y_{\text{aux}}^i\) is assigned \(\{0, 1, g\}\) for \(i \in \{W - 1, \ldots, 0\}\), and this was discussed in detail in Section 5.1.

We prove \(\Leftarrow\) by induction on the scanning order, from the MSB to the LSB, that the polynomials are set to zero iff the auxiliary variables are assigned according to Equation 5.8. This proof, together with the fact that \(Y_{\text{aux}}^0\) is the comparison result (see last constraint in the ACIP), shows that \((\vec{X}_{\text{arg}1}, \vec{X}_{\text{arg}2}, X'_{\text{output}}) \in R_{\text{compare}}\).

**Base case** \(i = W - 1\):

- If \(x_{\text{arg}1}^{W-1} = x_{\text{arg}2}^{W-1}\), the polynomial \(p_{\text{compare}}^{W-1}\) equals

  $$Y_{\text{aux}}^{W-1} + g.$$  

  This polynomial is set to zero iff \(Y_{\text{aux}}^{W-1} = g\).

- If \(x_{\text{arg}1}^{W-1} \neq x_{\text{arg}2}^{W-1}\), the polynomial \(p_{\text{compare}}^{W-1}\) equals

  $$Y_{\text{aux}}^{W-1} + (X_{\text{arg}1}^{W-1} \cdot (X_{\text{arg}2}^{W-1} + 1)).$$

  This polynomial is satisfied iff \(Y_{\text{aux}}^{W-1}\) is assigned 1 when \(x_{\text{arg}1}^{W-1} = 1 \land x_{\text{arg}2}^{W-1} = 0\).
and is assigned 0 when \( x_{\text{arg}_1}^{W-1} = 0 \wedge x_{\text{arg}_2}^{W-1} = 1 \).

These two cases imply that if the first constraint is satisfied, \( Y_{\text{aux}}^{W-1} \) is assigned the values according to Equation 5.8.

Assume the correctness of the claim for \( 1 < i + 1 < W - 1 \) and let’s prove it for \( i \).

By the induction assumption, \( Y_{\text{aux}}^{i+1} \) is assigned 0, 1 or \( g \), according to Equation 5.8.

- If \( Y_{\text{aux}}^{i+1} \) is assigned 1, \( p^i_{\text{compare}} \) equals

\[
Y_{\text{aux}}^i + 1,
\]

which is clearly satisfied iff \( Y_{\text{aux}}^i \) is assigned 1.

- If \( Y_{\text{aux}}^{i+1} \) is assigned 0, \( p^i_{\text{compare}} = Y_{\text{aux}}^i \), which is satisfied iff \( Y_{\text{aux}}^i \) is assigned zero.

- If \( Y_{\text{aux}}^{i+1} \) is assigned \( g \), \( p^i_{\text{compare}} \) equals

\[
Y_{\text{aux}}^i + g \cdot (X_{\text{arg}_1}^i + X_{\text{arg}_2}^i + 1) + X_{\text{arg}_1}^i \cdot (X_{\text{arg}_2}^i + 1)
\]

If \( x_{\text{arg}_1}^i = x_{\text{arg}_2}^i \), the polynomial is satisfied iff \( Y_{\text{aux}}^i \) is assigned \( g \).

If \( x_{\text{arg}_1}^i = 1 \wedge x_{\text{arg}_2}^i = 0 \), the polynomial is satisfied iff \( Y_{\text{aux}}^i \) is assigned 1. Otherwise, it is satisfied iff \( Y_{\text{aux}}^i \) is assigned 0.

We have shown that if the constraint system is satisfied, then each \( Y_{\text{aux}}^i \) must be assigned values according to Equation 5.8.

**Note.** One can notice that the first set of constraints, type checking for \( Y_{\text{aux}}^i \), is redundant. If \( p^i_{\text{compare}} \) constraints are satisfied, \( Y_{\text{aux}}^i \) variables are assigned 0, 1 or \( g \) (see the proof above). The last constraint is also redundant, because \( Y_{\text{aux}}^0 \) can be replaced by \( X'_{\text{output}} \) in \( p^0_{\text{compare}} \).

Until now, we have focused on comparing unsigned integers. In order to expand the \( \text{compare} \) functionality to comparison of signed integers, one should add a constraint that handles the sign of the numbers. If both numbers are non-negative or both are negative, the comparison is executed in the same way as an unsigned integers comparison (because we use two’s complement). If one integer is non-negative and the second is negative, the result of the comparison is clear from the most significant bit. There are several different ways to write a constraint that handles integers’ signs. One option is to replace \( p^0_{\text{compare}} \) with the next polynomial if a signed comparison is required:

\[
Y_{\text{aux}}^0 + (p^0_{\text{compare}} + Y_{\text{aux}}^0) \cdot (X_{\text{arg}_1}^{W-1} + X_{\text{arg}_2}^{W-1} + 1) + (1 + X_{\text{arg}_1}^{W-1}) \cdot X_{\text{arg}_2}^{W-1}
\]

One can notice that if \( x_{\text{arg}_1}^{W-1} = x_{\text{arg}_2}^{W-1} \), this polynomial equals \( p^0_{\text{compare}} \). Otherwise, \( x_{\text{arg}_1}^{W-1} \neq x_{\text{arg}_2}^{W-1} \) and the polynomial equals,

\[
Y_{\text{aux}}^0 + (1 + X_{\text{arg}_1}^{W-1}) \cdot X_{\text{arg}_2}^{W-1}
\]
which verifies that $Y^0_{aux}$ is assigned 0 if $x_{arg_1}$ is negative, and 1 if $x_{arg_2}$ is negative. Using this polynomial slightly increases the degree. In order to avoid the degree change, one can define a similar constraint by using $p^{W-1}_{compare}$ instead of $p^0_{compare}$.

To distinguish between comparison of signed and unsigned integers, we denote by $C^s_{compare}$ the former type of comparison and $C^u_{compare}$ the latter type.

**multiplication**

#auxiliary variables: $2W - 1$, max degree: 4, #polynomials: $4W$

Multiplication receives two additive unpacked representations of unsigned integers $arg_1 = a = (a_0, a_1, \ldots, a_{W-1})$ and $arg_2 = b = (b_0, b_1, \ldots, b_{W-1})$ and verifies that the output result equals the field element $g^{a \cdot b}$. We define the next constraint system:

$$
C_{multiplication}(\vec{X}_{arg_1}, \vec{X}_{arg_2}, \vec{X'}_{output}, \vec{Y}_{aux}, \vec{Y}_{aux2}) =
C_{multiplicative packing}(\vec{Y}_{arg_1}, g^a, \vec{Y}_{aux}) \cup
\{(X_{arg_2} \cdot (X_{arg_2} + 1))|0 \leq i < W\} \cup
\{p^i_{mul}|0 \leq i < W\} \cup
\{X'_{output} + Y^0_{aux2}\}
$$

where:

$$
p^{W-1}_{mul} = Y^{W-1}_{aux2} + Y^a \cdot Y^{W-1}_{arg_2} + Y^a + 1
$$

$$
p^i_{mul} = Y^i_{aux2} + Y^{i+1}_{aux2} \cdot Y^i_{aux2} \cdot (Y^a \cdot Y^i_{arg_2} + Y^i_{arg_2} + 1) \quad i < W - 1.
$$

**Note.** (1) We denote the $Y^0_{aux}$ variable by the name $Y^a$, because this name is more expressive of the variable’s “role”.

(2) The last constraint is redundant, and in order to save a variable, $Y^0_{aux2}$ can be replaced by $X'_{output}$ in $p^0_{mul}$.

The essence of this constraint system lies in $p_{mul}$ polynomials. Let’s start with a practical example. Denote $\mathbb{F}_2$ an algebraic field and let $a, b = 10$ be integers represented by 4 bits such that $|\mathbb{F}_{2^m}| > 2^8$. The constraint system should verify that $g^{result} = g^{10-b}$:

$$
(g^a)^{\sum_{i=0}^{2^3} b_i \cdot 2^i} = (g^a)^{3 \cdot (g^a)^{2^1}} = g^{8a} \cdot g^{2a} = g^{10a}
$$

$$
p^3_{mul} : y^3_{aux2} = 1 \cdot g^a + 1 + 1 = g^a
$$

$$
p^2_{mul} : y^2_{aux2} = g^a \cdot g^a \cdot (g^a \cdot 0 + 0 + 1) = g^{2a}
$$

$$
p^1_{mul} : y^1_{aux2} = g^{2a} \cdot g^a \cdot (g^a \cdot 1 + 1 + 1) = g^{5a}
$$

$$
p^0_{mul} : y^0_{aux2} = g^{5a} \cdot g^{5a} \cdot (g^a \cdot 0 + 0 + 1) = g^{10a}
$$

44
Lemma 5.3.3. Assuming \( Y_{x^a} \) is assigned \( g^a \) and \( \hat{X}_{arg_2} \) is the binary representation of the second argument, \( p_{mul} \) constraints are satisfied iff \( Y_{aux_2}^i \) is assigned \( (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1) \cdot \prod_{j=i+1}^{W-1} (g^a)^{2^{j-i}} \).

Proof. We prove this lemma by induction on the \( i \)th variable from \( i = W - 1 \) to 0.  

base case \( i = W - 1 \): \( p_{mul}^{W-1} \) is satisfied iff  

\[
y_{aux_2}^{W-1} = g^a \cdot x_{arg_2}^{W-1} + x_{arg_2}^{W-1} + 1,
\]

which, clearly, holds for the lemma.  

Let’s assume the correctness of the lemma for \( W - 1 \geq i + 1 \geq 1 \) and prove it for \( i \).  

From the assumption, we can conclude that \( p_{mul}^{i+1} \) is satisfied iff  

\[
y_{aux_2}^{i+1} = (g^a \cdot x_{arg_2}^{i+1} + x_{arg_2}^{i+1} + 1) \cdot \prod_{j=i+2}^{W-1} (g^a)^{2^{j-i-1}}.
\]

\( p_{mul}^i \) is satisfied iff  

\[
Y_{aux_2}^i = Y_{aux_2}^{i+1} \cdot Y_{aux_2}^{i+1} \cdot (Y_{g^a} \cdot X_{arg_2}^i + X_{arg_2}^i + 1);
\]

therefore,  

\[
y_{aux_2}^i = y_{aux_2}^{i+1} \cdot y_{aux_2}^{i+1} \cdot (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1)
\]

\[= (g^a \cdot x_{arg_2}^{i+1} + x_{arg_2}^{i+1} + 1)^2 \cdot \left( \prod_{j=i+2}^{W-1} (g^a)^{2^{j-i-1}} \right) \cdot (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1) \]

\[= (g^a \cdot x_{arg_2}^{i+1} + x_{arg_2}^{i+1} + 1)^2 \cdot \left( \prod_{j=i+2}^{W-1} (g^a)^{2^{j-i}} \right) \cdot (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1) \]

If \( x_{arg_2}^i = 0 \), \( A \) equals 1, and the product’s start index can be \( i + 1 \), because the condition in the product, \( x_{arg_2}^{i+1} = 1 \), does not hold. Therefore, decreasing the index does not change the multiplication result. Otherwise, \( x_{arg_2}^{i+1} = 1 \)

\[
y_{aux_2}^i = g^{2a} \cdot \left( \prod_{j=i+2}^{W-1} (g^a)^{2^{j-i}} \right) \cdot (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1) =
\]

\[= (g^a)^{2^{i+1-i}} \cdot \left( \prod_{j=i+2}^{W-1} (g^a)^{2^{j-i}} \right) \cdot (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1) =
\]

\[= \left( \prod_{j=i+1}^{W-1} (g^a)^{2^{j-i}} \right) \cdot (g^a \cdot x_{arg_2}^i + x_{arg_2}^i + 1) \]

In both cases, the claim holds.  

\[\square\]
From the induction we get that

\[
y_{\text{aux}}^0 = \left( \prod_{j=1}^{W-1} (g^a)^{2^j} \right) \cdot (g^a \cdot x \cdot x_{\text{arg}}^0 + x_{\text{arg}}^0 + 1)
\]

\[
= (g^a)^{\sum_{j=0}^{W-1} x_{\text{arg}}^2 \cdot 2^j} \cdot (g^a \cdot x_{\text{arg}}^0 + x_{\text{arg}}^0 + 1)
\]

\[
= (g^a)^{\sum_{j=0}^{W-1} x_{\text{arg}}^2 \cdot 2^j} = (g^a)^b = (g^a)^{b^2},
\]

which is exactly the desired result.

**Signed Representation**  The modification to signed integers is pretty straightforward.

We use the signed representation of Definition 5.2.3 and apply a similar technique to modify the constraint in this constraint system. The only constraint that is modified is \( p_{\text{mul}}^{W-1} \), which we switch to

\[
\{ Y_{g^a} \cdot Y_{g^a} + 1 \} \cup \{ Y_{\text{aux}}^{W-1} + Y_{g^a} \cdot X_{\text{arg}}^{W-1} + X_{\text{arg}}^{W-1} + 1 \}.
\]

The first constraint is satisfied if-and-only-if \( Y_{g^a} \) is assigned with \( g^a \). The second constraint is satisfied:

- If \( x_{\text{arg}}^{W-1} = 1 \), the second argument represents a negative integer; therefore, the polynomial is satisfied iff \( Y_{\text{aux}}^{W-1} \) is assigned \( g^a \). We can show, by induction, that \( y_{\text{aux}}^0 = g^a \cdot |b| \), in the same way we proved the unsigned case.

- If \( x_{\text{arg}}^{W-1} = 0 \), in order for the constraints to be satisfied, the assignments should equal the unsigned case.

### 5.3.2 Binary Opcodes

**xor**

- **#auxiliary variables:** 1, **max degree:** 2, **#polynomials:** 3

The xor operation treats integers as a series of bits and executes a bitwise exclusive-or on the inputs, and stores the result in the output operand. In addition, it sets the flag variable to 1 if the result of the xor is 0; otherwise it is set to 1. In other words, a \((2,2)\) – xor opcode can be defined by the relation

\[
R_{\text{xor}}(x_{\text{arg}1}, x_{\text{arg}2}, x_{\text{output}}, x_{\text{flag}}) = \\
\{(x_{\text{arg}1}, x_{\text{arg}2}, x_{\text{arg}1} \oplus x_{\text{arg}2}, 0)|x_{\text{arg}1} \neq x_{\text{arg}2}\} \cup \\
\{(x_{\text{arg}1}, x_{\text{arg}2}, x_{\text{arg}1} \oplus x_{\text{arg}2}, 1)|x_{\text{arg}1} = x_{\text{arg}2}\}
\]

Due to the fact that in addition in \( \mathbb{F}_{2^m} \) and xor are essentially the same, a simple
ACIP is defined.

\[
C_{\text{xor}}(X_{\text{arg}_1}, X_{\text{arg}_2}, X'_{\text{output}}, X'_{\text{flag}}, Y) = \\
\{ X'_{\text{output}} + X_{\text{arg}_1} + X_{\text{arg}_2} \} \cup \quad (5.12)
\]

and

#auxiliary variables: \(2W + 1\), max degree: 2, #polynomials: \(2W + 5\)

and gets two numbers and executes a bitwise and. If both of the bits are 1, the resulting bit is set to 1; otherwise, it is set to 0. The flag is assigned 1 if the output of the operation is 0; otherwise it is to 0. One can describe this operand as a \((2, 2)\) – and opcode in the following way:

\[
R_{\text{and}}(X_{\text{arg}_1}, X_{\text{arg}_2}, X'_{\text{output}}, X'_{\text{flag}}) = \\
\{ (X_{\text{arg}_1}, X_{\text{arg}_2}, X_{\text{arg}_1} & X_{\text{arg}_2}) \} \cup \quad (5.13)
\]

As opposed to xor, it is unclear how to write a succinct constraint system that exploits the properties of fields of characteristic 2. Hence, the ACIP we propose mimics an and on pairs of corresponding bits.

\[
C_{\text{and}}(X_{\text{arg}_1}, X_{\text{arg}_2}, X'_{\text{output}}, X'_{\text{flag}}, Y_{\text{arg}_1}, Y_{\text{arg}_2}, Y'_{\text{flag}}) = \\
C_{\text{additive unpacking}}(X_{\text{arg}_1}, Y_{\text{arg}_1}) \cup \\
C_{\text{additive unpacking}}(X_{\text{arg}_2}, Y_{\text{arg}_2}) \cup \\
\{ X'_{\text{output}} + \sum_{i=0}^{W-1} \beta_i \cdot Y^i_{\text{arg}_1} \cdot Y^i_{\text{arg}_2} \} \cup \\
C_{\text{Indicator}}(X'_{\text{output}}, X'_{\text{flag}}, Y'_{\text{flag}}) \\
\]

where \(\{\beta_0, \ldots, \beta_{m-1}\}\) is the standard basis for \(\mathbb{F}_{2^m}\) over \(\mathbb{F}_2\).

**Lemma 5.3.4.** \((x_{\text{arg}_1}, x_{\text{arg}_2}, x'_{\text{output}}, x'_{\text{flag}}) \in R_{\text{and}} \iff \exists(y_{\text{arg}_1}, y_{\text{arg}_2}, y'_{\text{output}}, y'_{\text{flag}}) \in \mathbb{F}_2^{3W+1} s.t C_{\text{and}}(x_{\text{arg}_1}, x_{\text{arg}_2}, x'_{\text{output}}, x'_{\text{flag}}, y_{\text{arg}_1}, y_{\text{arg}_2}, y'_{\text{flag}})\) is satisfied.

**Proof.** \(\Leftarrow\) As have been shown earlier, \(C_{\text{additive unpacking}}\) validates the conversion between the additive packed mode and additive unpacked mode. If these constraints are satisfied, then,

\[
y^i_{\text{arg}_j} = 1 \iff \text{the } i\text{th bit of } x_{\text{arg}_j} \text{ is 1,}
\]

for \(j \in \{1, 2\}\). In other words, \(Y_{\text{arg}_1}\) and \(Y_{\text{arg}_2}\) are the variables matching the binary representation of \(x_{\text{arg}_1}\) and \(x_{\text{arg}_2}\), respectively. The third constraint is set to zero iff \(X'_{\text{output}}\) is assigned with \(\sum_{i=0}^{W-1} \beta_i \cdot y^i_{\text{arg}_1} \cdot y^i_{\text{arg}_2}\). If either \(y^i_{\text{arg}_1}\) or \(y^i_{\text{arg}_2}\) is 0, then the
polynomial \( y^i_{arg_1} \cdot y^i_{arg_2} \) equals to zero, otherwise it equals to one. Formally,

\[
    y^i_{arg_1} \cdot y^i_{arg_2} \text{ equals to 1 } \iff \text{ the } i^{th} \text{ bit in } x_{arg_1} \cup x_{arg_2} \text{ equals 1,}
\]

for \( 0 \leq i \leq W - 1 \). Therefore, the third constraint is satisfied iff the variable which represents the output, \( X'_{output} \), is assigned with the additive packed representation of \( x_{arg_1} \cup x_{arg_2} \). The fourth constraint was discussed in detail in Lemma 5.3.1.

\[\Rightarrow\] It is clear that if \( Y_{arg_1}, Y_{arg_2} \) are assigned the binary representations of \( x_{arg_1}, x_{arg_2} \), respectively, and \( X'_{output} \) is assigned with the additive representation of \( x_{arg_1} \cup x_{arg_2} \), then the constraint system is satisfied. 

\[\Box\]

or

**#auxiliary variables:** \( 2W + 1 \), **max degree:** 2, **#polynomials:** \( 2W + 5 \)

Similarly to **and**, **or** executes a bitwise operation on corresponding bits. Contrary to **and**, the bitwise operation in this case is a logical **or**. If both of the bits are 0, the resulting bit is set to 0; otherwise, it is set to 1. We can define the \((2, 2) - \text{or op code by the relation,} \]

\[
    R_{or}(x_{arg_1}, x_{arg_2}, X'_{output}, X'_{flag}) = \\
    \{(x_{arg_1}, x_{arg_2}, x_{arg_1} | x_{arg_2}, 0) | x_{arg_1} \neq 0\} \cup \{(x_{arg_1}, x_{arg_2}, x_{arg_1} | x_{arg_2}, 1) | x_{arg_1} = 0\}
\]

(5.15)

Due to the similarity between **and** and **or**, only the third constraint should be changed.

\[
    C_{or}(X_{arg_1}, X_{arg_2}, X'_{output}, X'_{flag}, Y_{arg_1}, Y_{arg_2}, Y'_{flag}) = \\
    C_{additive unpacking}(X_{arg_1}, Y_{arg_1}) \cup \\
    C_{additive unpacking}(X_{arg_2}, Y_{arg_2}) \cup \\
    \left\{X'_{output} + \sum_{i=0}^{W-1} \beta_i \cdot (1 + (1 + Y^i_{arg_1}) \cdot (1 + Y^i_{arg_2}))\right\} \cup \\
    C_{Indicator_0}(X'_{output}, X'_{flag}, Y'_{flag})
\]

(5.16)

The third constraint is an implementation of De Morgan’s laws, and it is satisfied iff \( X'_{output} \) is assigned with \( \sum_{i=0}^{W-1} \beta_i \cdot (1 + (1 + y^i_{arg_1}) \cdot (1 + y^i_{arg_2})) \). Obviously,

\[
    1 + (1 + y^i_{arg_1}) \cdot (1 + y^i_{arg_2}) \text{ equals 1 } \iff \text{ the } i^{th} \text{ bit of } x_{arg_1} | x_{arg_2} \text{ is 1}
\]

for each \( 0 \leq i \leq W - 1 \). Due to additive unpacking constraints, the \( i^{th} \) bit of \( x_{arg_j} \) for \( j \in \{1, 2\} \) is 1 if \( y^i_{arg_j} = 1 \). If the \( i^{th} \) bit of either \( x_{arg_1} \) or \( x_{arg_2} \) is 1, then the polynomial \( 1 + (1 + y^i_{arg_1}) \cdot (1 + y^i_{arg_2}) \) equals to 1. Otherwise, the \( i^{th} \) bit of both \( x_{arg_1} \) and \( x_{arg_2} \) is 0, therefore, \( 1 + (1 + y^i_{arg_1}) \cdot (1 + y^i_{arg_2}) \) equals to 0. Clearly, the third polynomial is satisfied iff \( X'_{output} \) is assigned with the additive representation of \( x_{arg_1} | x_{arg_2} \).
not

#auxiliary variables: 1, max degree: 2, #polynomials: 3

not is an unary operation that inverts each bit in the number; 0 becomes 1 and 1 becomes 0. As other bitwise operands, the flag is set to 1 iff the output equals 0. One can define a \((1, 2)\) – not opcode by the relation,

\[
R_{\text{not}}(x_{\text{arg}_1}, x'_{\text{output}}, x'_{\text{flag}}) = \{(x_{\text{arg}_1}, \sim x_{\text{arg}_1}, 0) | \sim x_{\text{arg}_1} \neq 0\} \cup \{(x_{\text{arg}_1}, \sim x_{\text{arg}_1}, 1) | \sim x_{\text{arg}_1} = 0\}. \tag{5.17}
\]

Let \(\mathcal{B} = \{\beta_0, \ldots, \beta_{m-1}\}\) be the standard basis for \(\mathbb{F}_2^m\) over \(\mathbb{F}_2\). We denote by \(e_{\text{ones}}\) the field element that equals to \(\sum_{i=0}^{W-1} \beta_i\). One can notice that the addition of a field element that is the additive representation of an integer of size \(W\) with \(e_{\text{ones}}\), basically “flips all the bits”. We define the next \(ACIP\) for the not opcode

\[
C_{\text{not}}(X_{\text{arg}_1}, X'_{\text{output}}, X'_{\text{flag}}, Y'_{\text{flag}}) = \{X'_{\text{output}} + X_{\text{arg}_1} + e_{\text{ones}}\} \cup C_{\text{Indicator}_0}(X'_{\text{output}}, X'_{\text{flag}}, Y'_{\text{flag}}). \tag{5.18}
\]

**Lemma 5.3.5.** \((x_{\text{arg}_1}, x'_{\text{output}}, x'_{\text{flag}}) \in R_{\text{not}} \iff \exists y'_{\text{flag}} \in \mathbb{F} \text{ s.t. } C_{\text{not}}(x_{\text{arg}_1}, x'_{\text{output}}, x'_{\text{flag}}, y'_{\text{flag}}) \text{ is satisfied}\)

**Proof.** In the \(ACIP\) described in Equation 5.6, we showed that \(x'_{\text{flag}}\) is 1 iff \(X'_{\text{output}}\) is assigned 0.

\(\Leftarrow\) The first polynomial is set to zero iff \(X'_{\text{output}}\) is assigned \(x_{\text{arg}_1} + e_{\text{ones}}\), which is an additive representation of \(\sim x_{\text{arg}_1}\). Therefore, \((x_{\text{arg}_1}, x'_{\text{output}}, x'_{\text{flag}}) \in R_{\text{not}}\)

\(\Rightarrow\) If \((x_{\text{arg}_1}, \sim x_{\text{arg}_1}, x'_{\text{flag}}) \in R_{\text{not}}, X'_{\text{output}}\) is assigned the algebraic translation of \(\sim x_{\text{arg}_1}\) which is \(x_{\text{arg}_1} + e_{\text{ones}}\). This assignment sets the first polynomial to zero. The second \(ACIP\) is set to zero according to Lemma 5.3.1.

\(\square\)

### 5.3.3 Integer opcodes

add

#auxiliary variables: \(4W - 1\), max degree: 2, #polynomials: \(4W + 3\)

Generally speaking, the add opcode gets two integers that are represented by \(W\) bits and adds them. There is, however, a subtlety one should notice. Adding two integers, which are represented by \(W\) bits, can result an integer that is represented by \(W + 1\) bits. Hence, the result of this opcode is the \(W\) least significant bits of the addition, and the flag is set to 1 if an overflow occurred and to 0 otherwise. We can define addition as
an \((2,2)\) \(−\) add opcode by the relation,

\[
\mathcal{R}_{\text{add}}(x_{\arg 1}, x_{\arg 2}, x'_{\text{output}}, x'_{\text{flag}}) = \{ (x_{\arg 1}, x_{\arg 2}, (x_{\arg 1} + x_{\arg 2})^0,...,x_{\arg 1}^{W-1}, (x_{\arg 1} + x_{\arg 2})^W) \}.
\]

(5.19)

The ACIP we propose basically verifies the “standard” addition algorithm, aligning the numbers vertically, and adding all the bits in the same column from the LSB. For example, if each register is represented by 3 bits and we want to add 5(101) and 3(011),

\[
\begin{array}{c c c}
\text{carry} & 1 & 1 \\
+ & 0 & 1 \\
\hline
& 1 & 0 \\
\end{array}
\]

The result of the addition is 0 and the flag is set to 1.

\[
\text{C}_{\text{add}}(X_{\arg 1}, X_{\arg 2}, X'_{\text{output}}, X'_{\text{flag}}, Y_{\arg 1}, Y_{\arg 2}, Y'_{\text{output}}, Y'_{\text{flag}}) = \text{C}_{\text{additive unpacking}}(X_{\arg 1}, Y_{\arg 1}) \cup \text{C}_{\text{additive unpacking}}(X_{\arg 2}, Y_{\arg 2}) \cup \{ p^i_{\text{output}} | 0 \leq i < W \} \cup \{ p^i_{\text{carry}} | 0 \leq i < W \} \cup \{ Y^W_{\text{carry}} + X'_{\text{flag}} \} \cup \text{C}_{\text{additive packing}}(Y'_{\text{output}}, X'_{\text{output}})
\]

(5.20)

where:

\[
p^0_{\text{output}} = Y^0_{\text{output}} + Y^0_{\arg 1} + Y^0_{\arg 2}
\]

\[
p^i_{\text{output}} = Y^i_{\text{output}} + Y^i_{\arg 1} + Y^i_{\arg 2} + Y^{i-1}_{\text{carry}} \quad i > 0
\]

and

\[
p^0_{\text{carry}} = Y^0_{\text{carry}} + Y^0_{\arg 1}, Y^0_{\arg 2}
\]

\[
p^i_{\text{carry}} = Y^i_{\text{carry}} + Y^i_{\arg 1}, Y^i_{\arg 2} + Y^{i-1}_{\text{carry}} + Y^0_{\arg 1}, Y^{i-1}_{\text{carry}} \quad i > 0
\]

Note. (1) The \(i\)th carry is aligned vertically with bits \((i + 1)\) of the input.

(2) The number of auxiliary variables is \(4W - 1\) instead of \(4 \cdot W\), because the fifth constraint is redundant. The same variable can be used for \(Y^{W-1}_{\text{carry}}\) and \(X'_{\text{flag}}\).

(3) There is no need to check that \(Y^i_{\text{output}}\) is assigned either 0 or 1 in the additive packing constraint system because the polynomials \(p^i_{\text{output}}\) and \(p^i_{\text{carry}}\) are set to zero iff this assignment holds.

According to the binary addition algorithm, the \(i\)th carry equals 1 iff at least two bits in the \(i - 1\) column equal 1, and the \(i\)th bit of the output equals 1 iff the number of 1’s in
the \(i\)th column is odd (in our case, either 1 or 3). We mimic this behavior over a field. Formally, the values of \(y^i_{\text{carry}}\) and \(y'^i_{\text{output}}\) can be defined according to the following equation:

\[
y^i_{\text{carry}} = \begin{cases} 
1 & (i = 0 \land y^0_{\text{arg1}} = 1 \land y^0_{\text{arg2}} = 1) \lor \\
(i > 0 \land y^i_{\text{arg1}} = 1 \land y^i_{\text{arg2}} = 1) \lor \\
(i > 0 \land y^i_{\text{arg1}} = 1 \land y^{i-1}_{\text{carry}} = 1) \lor \\
(i > 0 \land y^i_{\text{arg2}} = 1 \land y^{i-1}_{\text{carry}} = 1) \\
0 & \text{otherwise}
\end{cases}
\]

(5.21)

\[
y'^i_{\text{output}} = \begin{cases} 
1 & (i = 0 \land (y^0_{\text{arg1}} + y^0_{\text{arg2}} \mod 2 = 1)) \lor \\
(i > 0 \land (y^i_{\text{arg1}} + y^i_{\text{arg2}} + y^{i-1}_{\text{carry}} \mod 2 = 1)) \\
0 & \text{otherwise}
\end{cases}
\]

In order not to be repetitive and redundant in our proofs, we focus on the essence of proving \((x_{\text{arg1}}, x_{\text{arg2}}, x'_{\text{output}}, x'_{\text{flag}}) \in \mathcal{R}_{\text{add}} \iff \exists y_{\text{arg1}}, y_{\text{arg2}}, y'_{\text{output}}, y_{\text{carry}} \in \mathbb{F}_4^{W-1} \) s.t. \(C_{\text{add}}(x_{\text{arg1}}, x_{\text{arg2}}, x'_{\text{output}}, x'_{\text{flag}}, y_{\text{arg1}}, y_{\text{arg2}}, y'_{\text{output}}, y_{\text{carry}})\) is satisfied, which is proving that \(y^i_{\text{carry}}\) and \(y'^i_{\text{output}}\) are assigned according to Equation 5.21.

**Lemma 5.3.6.** Assuming that \(\bar{y}_{\text{arg1}}, \bar{y}_{\text{arg2}}\) are assigned the additive unpacked representation of \(x_{\text{arg1}}, x_{\text{arg2}}\), then \(p^i_{\text{output}}\) and \(p^i_{\text{carry}}\) are satisfied iff \(y^i_{\text{carry}}\) and \(y'^i_{\text{output}}\) are assigned according to Equation 5.21.

**Proof.** We will prove the lemma by induction on the \(i\)th variable, where \(i = 0, \ldots, W-1\).

**Base case** \(i = 0\): The polynomial \(p^0_{\text{output}}\) is satisfied iff

\[
y^0_{\text{output}} = y^0_{\text{arg1}} + y^0_{\text{arg2}}
\]

\(y^0_{\text{output}}\) is assigned 1 iff only one of the variables, either \(y^0_{\text{arg1}}\) or \(y^0_{\text{arg2}}\), is assigned 1. Clearly, this assignment matches Definition in 5.21. In the same way, \(p^i_{\text{carry}}\) is satisfied iff

\[
y^i_{\text{carry}} = y^0_{\text{arg1}} \cdot y^0_{\text{arg2}}
\]

\(y^0_{\text{carry}}\) is assigned 1 if and only if both \(y^0_{\text{arg1}}\) and \(y^0_{\text{arg2}}\) are assigned 1. In other words, \(y^0_{\text{carry}} = 1\) iff \(y^0_{\text{arg1}} = 1 \land y^0_{\text{arg2}} = 1\).

Let’s assume the correctness for \(i \geq 0\) and prove it for \(i + 1\).

\(p^{i+1}_{\text{output}}\) is set to zero iff

\[
y'^{i+1}_{\text{output}} = y^{i+1}_{\text{arg1}} + y^{i+1}_{\text{arg2}} + y'^i_{\text{carry}}
\]

Because these variables are assigned either 0 or 1 (induction assumption and lemma’s data), and the polynomial is defined over a field of characteristic 2, \(y'^{i+1}_{\text{output}}\) equals
1 iff an odd number of variables are assigned with 1 (either 1 or 3 of the variables \( Y_{arg_1}, Y_{arg_2}, Y_{carry} \) are assigned 1). This proves that \( p_{output}^{i+1} \) is set to zero iff \( Y_{output}^{i+1} \) is assigned according to Equation 5.21.

\[ Y_{carry}^{i+1} = Y_{arg_1}^{i+1} \cdot Y_{arg_2} + Y_{arg_1}^{i+1} \cdot Y_{carry} + Y_{arg_2}^{i+1} \cdot Y_{carry} \]

The truth table (Table 5.2),

<table>
<thead>
<tr>
<th>( y_{arg_1}^{i+1} )</th>
<th>( y_{arg_2}^{i+1} )</th>
<th>( y_{carry}^i )</th>
<th>( y_{carry}^{i+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: Binary addition - truth table

shows that the \((i + 1)\) bit of the carry equals 1 iff at least two of the three variables, \( Y_{arg_1}^{i+1}, Y_{arg_2}^{i+1}, Y_{carry}^i \), are assigned 1; otherwise it equals 0. This also shows that the polynomial is set to zero iff \( Y_{carry}^{i+1} \) is assigned according to Equation 5.21.

\[ \text{Lemma (5.3.6), the additive unpacking ACIP and the additive packing ACIP show that } (x_{arg_1}, x_{arg_2}, x'_{output}, x'_{output}) \in (2, 2) \Rightarrow \text{add opcode iff } \exists (y_{arg_1}, y_{arg_2}, y_{output}, y_{carry}) \in \mathbb{F}^{4W} \text{ s.t } C_{\text{add}}(x_{arg_1}, x_{arg_2}, x'_{output}, x'_{flag}, y_{arg_1}, y_{arg_2}, y_{output}, y_{carry}) \text{ is set to zero.} \]

**Sub**

*auxiliary variables: \( 4W - 1 \), **max degree: 2, **polynomials: \( 4W + 3 \)*

Sub and add share a special relation. Informally, the constraints in add are satisfied iff

\[
(x_{arg_1} + x_{arg_2})^{0 \ldots W-1} = \text{Result} \\
(x_{arg_1} + x_{arg_2})^W = \text{Flag}
\]

In sub, the objective is to define a constraint system that is satisfied iff the result equals the \( W \) least significant bits of \( x_{arg_1} - x_{arg_2} \), and the flag is set to 1 if there was a borrow; otherwise it is set to 0. Due to the similarity of the opcodes, we can define the subtraction constraint system with the help of addition constraint system. We actually
change the roles of the summands and the result becomes:

\[
C_{\text{sub}}(Y_{\text{arg}1}, Y_{\text{arg}2}, X'_\text{output}, X'_\text{flag}, Y_{\text{arg}1}, Y_{\text{arg}2}, Y'_\text{output}, Y'_\text{carry}) =
C_{\text{add}}(Y_{\text{arg}2}, X'_\text{output}, Y_{\text{arg}1}, X'_\text{flag}, Y_{\text{arg}2}, Y'_\text{output}, Y_{\text{arg}1}, Y'_\text{carry})
\] (5.22)

One can easily verify that the this constraint system is satisfied iff the integer subtraction is executed correctly.

**multiplication opcodes**

In many architectures, there are several multiplication opcodes, i.e., mull, umullh, smullh, etc. The key differences between these opcodes are: (1) whether the multiplication is performed on signed or unsigned integers and (2) which bits represent the result (recall that multiplying two integers that are represented by \(W\) can result in an integer of length \(2 \cdot W\)).

There are several ways to define a constraint system for multiplication opcodes. One way is by continuing to imitate “school” methods. Although in addition and subtraction, these methods helped to define a succinct constraint system, long multiplication adds redundant constraints and variables. In this technique, we would have \(W\) shifted additions of \(W\) length numbers, which implies that the number of polynomials and the number of auxiliary variables would be at least \(W^2\). In order to avoid this redundancy, we decided to use the following property,

\[
(g^a)^b = g^{a \cdot b} = g^{\text{output}}
\]

\[
(g^a)\sum_{i=0}^{W-1} (b_i \cdot 2^i) = \prod_{i=0 \land b_i = 1}^{W-1} (g^a)^{2^i} = g^{\text{output}}
\]

where \(a, b\) are integers represented by \(W\) bits and \(g\) is the generator of the multiplicative group. We assume that the multiplicative group is bigger than \(2^{2W}\). Without this assumption, there may be an overlap leading to \(a \cdot b \neq \text{output}\).

**mull** #auxiliary variables: \(8W\), max degree: 4, #polynomials: \(8W + 6\)

The **mull** opcode gets two unsigned integers and returns the \(W\) least significant bits as a result. It also sets the flag to 1 if the \(W\) most significant bits are all zeros; otherwise it sets the flag to 0. One can define this relation by \((2, 2) - \text{mull opcode}\)

\[
R_{\text{mull}}(x_{\text{arg}1}, x_{\text{arg}2}, x'_\text{output}, x'_\text{flag}) =
\{ x_{\text{arg}1}, x_{\text{arg}2}, (x_{\text{arg}1} \cdot x_{\text{arg}2})_{W-1, 0}, 1 | (x_{\text{arg}1} \cdot x_{\text{arg}2})_{W, 2W-1} = 0 \} \cup
\{ x_{\text{arg}1}, x_{\text{arg}2}, (x_{\text{arg}1} \cdot x_{\text{arg}2})_{W-1, 0}, 0 | (x_{\text{arg}1} \cdot x_{\text{arg}2})_{W, 2W-1} \neq 0 \}
\] (5.23)

Based on the above discussion, one can define the next the ACIP presented in Equation 5.24.
\[ C_{\text{mull}}(X_{\text{arg}1}, X_{\text{arg}2}, X'_{\text{output}}, X'_{\text{flag}}, Y_{\text{arg}1}, Y_{\text{arg}2}, Y_{\text{aux}}, Y_{\text{aux}2}, Y'_{\text{output}}) =
\]
\[ C_{\text{additive unpacking}}(X_{\text{arg}1}, Y_{\text{arg}1}) \cup
C_{\text{additive unpacking}}(X_{\text{arg}2}, Y_{\text{arg}2}) \cup
C_{\text{multiplication}}(Y_{\text{arg}1}, Y_{\text{arg}2}, Y'_{\text{aux}}, Y_{\text{aux}}) \cup
C_{\text{multiplicative unpacking}}(Y'_{\text{aux}}, Y'_{\text{output}}, Y_{\text{aux}2}) \cup
C_{\text{additive packing}}(Y'_{\text{output}}, X'_{\text{output}}) \]
\[ \left\{ X'_{\text{flag}} + \prod_{i=W}^{2W-1} (Y'_{\text{output}} + 1) \right\} \]

\[ \text{(5.24)} \]

Note. In the first three constraint systems, in mull’s ACIP, there are type checking constraints that force \( X_{\text{arg}1} \) and \( X_{\text{arg}2} \) to be assigned either 0 or 1. There is no need for these multiple constraints. Therefore, in this opcode, the type checking constraints either in additive unpacking or multiplication can be neglected.

In mull’s ACIP, the only constraint that still has not been discussed is the last one, which is responsible for flag validity. In the presented polynomial, if any of \( y'_{\text{output}} \) equals 1 (for \( i = W, \ldots, 2 \cdot W - 1 \)), then the product equals 0 and hence the flag must be set to 0. Otherwise, all \( y'_{\text{output}} \)’s are equal to zero, and the polynomial is satisfied iff the flag is set to 1. There are several ways to write a polynomial that captures the flag validity, but there is a trade-off between the number of constraints, the degree of the polynomials and the number of auxiliary variables. In the above constraint, the degree of the polynomial is high, namely, \( W \), and our objective is to reduce it. This polynomial can be replaced by,

\[ C_{\text{additive packing}}(Y'_{\text{result}}^{2W-2} \cdots 1, Y_{\text{flagAux}}) \cup
C_{\text{Indicator}}(Y_{\text{flagAux}}, X'_{\text{flag}}, Y_{\text{inverse}}) \]

Although, two constraints and two auxiliary variables are added, the degree of these constraints is reduced from \( W \) to 2.

\[ \text{umulh } \# \text{auxiliary variables: } 8W, \text{ max degree: } 4, \# \text{polynomials: } 8W + 6 \]

The only difference between \( \text{umulh} \) and \( \text{mull} \) is the bits that represent the result. In \( \text{mull} \), the result of the multiplication is the \( W \) least significant bits as opposed to \( \text{umulh} \) where the result is represented by the \( W \) most significant bits. Due to the similarity, the ACIP differs only in the \( C_{\text{additive packing}} \) constraint. In \( \text{umulh} \), the fifth set of constraints is

\[ C_{\text{additive packing}}(X'_{\text{output}}, Y'_{\text{result}}^{2W-2}) \]

54
**smulh**  smulh opcode gets two signed integers and returns, \( W \) most significant bits of the multiplication. *smulh’s* constraint system is essentially equivalent to *umulh’s*. The only difference between these constraint systems is the mode of multiplicative packing, unpacking and multiplication work. In *mul* and *umulh*, the sequence of bits represents unsigned integers, while in *smulh* they represent signed integers. In both multiplication opcode and multiplicative packing, we showed how to extend the constraint systems to verify signed representations as well. Therefore, we can use the same *ACIP* presented in *umulh*

*smulh* is an excellent example to demonstrate why our *ACIP* is superior and more simple than the “school” method. Not only are there \( W \) shifted additions between the \( W \) length numbers in the school method, but also special constraints are needed to verify the multiplication of the signed integers. Without adaptation, a signed multiplication may return a wrong result. For example,

\[
\begin{array}{c|c|c|c}
\text{(-1)} & 1 & 1 & 1 \\
\times & & & \\
\text{(-2)} & 1 & 1 & 0 \\
\hline
0 & 0 & 0 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\hline
\text{(-22)} & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
\]

One option for solving this problem is to double the precision of the integers before multiplication. In the example above, instead of representing an integer with 3 bits, we would represent it with 6. There is no doubt that this method is inefficient. By implementing this technique, we increase both the number of additions and their length. In other words, the constraint system would contain more polynomials, and would be defined over an increased number of variables. There are other methods to implement signed multiplication, however, they are less intuitive than doubling the precision and they use other opcodes such as *shift* and *add*, which do not improve their efficiency in comparison to our method.

**udiv**

*auxiliary variables*: \( 9W + 1 \), **max degree**: 4, **polynomials**: \( 9W + 7 \)

Given two integers \( a, b \neq 0 \), \( a \) can be uniquely represented as

\[ a = b \cdot q + r \]

where \( q \in \mathbb{N} \) (quotient) and \( 0 \leq r < b \) (remainder). *Udiv* is an opcode that gets two unsigned integers and sets the flag and the result under the following conditions:

- If the second argument equals zero, the result is 0 and the flag is set to 1.
• Otherwise, the flag is set to 0 and the result is set to the integer division of the first argument by the second (or quotient as defined above).

We can define this relation as a $(2, 2) - \text{udiv}$ opcode in the following way:

$$
\mathcal{R}_{\text{udiv}}(x_{\text{arg}1}, x_{\text{arg}2}, x'_{\text{output}}, x'_{\text{flag}}) = \\
\{ x_{\text{arg}1}, x_{\text{arg}2}, 0, 1 | x_{\text{arg}2} = 0 \} \cup \\
\{ x_{\text{arg}1}, x_{\text{arg}2}, q, 0 | x_{\text{arg}2} \neq 0, x_{\text{arg}1} = q \cdot x_{\text{arg}2} + r, r < x_{\text{arg}2}, q, r \in \mathbb{N} \}
$$

(5.25)

We define a constraint system that validates the equation above. Our polynomials verify that

1. When $b = x_{\text{arg}2} \neq 0$, $g^a = g^q b \cdot g^r$ such that
   
   (a) $a = x_{\text{arg}1}$ and $q = x'_{\text{output}}$.
   
   (b) $0 \leq r < b$, with comparison $ACIP$.
   
   (c) $q \cdot b \mapsto g^q b$ with multiplication $ACIP$.
   
   (d) $a \mapsto g^a$ and $r \mapsto g^r$ with multiplicative (un)packing $ACIP$.
   
   (e) $x'_{\text{flag}} = 0$.

2. $b = x_{\text{arg}2} = 0$ then $x'_{\text{flag}} = 1$ and $x'_{\text{output}} = 0$.

We propose the next $ACIP$:

$$
\mathcal{C}_{\text{udiv}}(X_{\text{arg}1}, X_{\text{arg}2}, X'_{\text{output}}, X'_{\text{flag}}, Y_{g^a}, Y_{g^r}, Y_{\text{inv}}, Y_{\text{arg}1}, Y_{\text{arg}2}, Y_{\text{aux}1}, Y_{\text{aux}2}, Y_{\text{aux}3}, Y_{\text{aux}4}, Y_{r}, Y'_{\text{output}}) = \\
\mathcal{C}_{\text{additive unpacking}}(X_{\text{arg}2}, Y_{\text{arg}2}) \cup \\
\mathcal{C}_{\text{additive unpacking}}(X'_{\text{output}}, Y'_{\text{output}}) \cup \\
\mathcal{C}_{\text{multiplication}}(Y_{\text{arg}2}, Y'_{\text{output}}, Y_{\text{aux}2}, Y_{\text{aux}1}, Y_{\text{aux}3}) \cup \\
\mathcal{C}_{\text{additive unpacking}}(X_{\text{arg}1}, Y_{\text{arg}1}) \cup \\
\mathcal{C}_{\text{multiplicative packing}}(Y_{r}, Y_{g^a}, Y_{\text{aux}3}) \cup \\
\mathcal{C}_{\text{multiplicative packing}}(Y_{r}, Y_{g^r}, Y_{\text{aux}4}) \cup \\
\mathcal{C}_{\text{compare}}(Y_{\text{arg}2}, Y_{r}, Y_{0_{aux5}}, Y_{\text{aux}5}) \cup \\
\{ X_{\text{arg}2} \cdot (X_{\text{arg}2} \cdot Y_{\text{inv}} + 1) \} \cup \\
\{ X_{\text{arg}2} \cdot (Y_{g^a} + Y_{0_{aux5}} \cdot Y_{g^r}) + (X_{\text{arg}2} \cdot Y_{\text{inv}} + 1) \cdot X'_{\text{output}} \} \cup \\
\{ X_{\text{arg}2} \cdot (1 + Y_{0_{aux5}}) \} \cup \\
\{ X'_{\text{flag}} + X_{\text{arg}2} \cdot Y_{\text{inv}} + 1 \}
$$

(5.26)

The first seven $ACIP$s above have already been discussed. Here we focus on the last four constraints. The first constraint in this group,

$$X_{\text{arg}2} \cdot (X_{\text{arg}2} \cdot Y_{\text{inv}} + 1),$$

56
is satisfied iff

- $x_{\text{arg}2} = 0$ or
- $x_{\text{arg}2} \neq 0$, then $\exists y_{\text{inv}} \in \mathbb{F}_{2^m}$ s.t. $x_{\text{arg}2} \cdot y_{\text{inv}} = 1$.

The second constraint,

$$X_{\text{arg}2} \cdot (Y_{g^e} + Y_{\text{aux}2}^0 \cdot Y_{gr}^e) + (X_{\text{arg}2} \cdot Y_{\text{inv}} + 1) \cdot X'_{\text{output}}.$$  

is satisfied iff

- $x_{\text{arg}2} = 0$, then the first argument is set to zero. Therefore, in order to satisfy the polynomial, $X'_{\text{output}}$ must be set to zero. This case captures the condition $x_{\text{arg}2} = 0 \rightarrow x'_{\text{output}} = 0$.
- $x_{\text{arg}2} \neq 0$, then the second argument is set to zero (because of the previous constraint). This polynomial is set to zero iff $Y_{g^e} = Y_{\text{aux}2}^0 \cdot Y_{gr}^e$. Recall that the first seven constraint systems verify that $Y_{g^e}$, $Y_{gr}^e$ and $Y_{\text{aux}2}^0$ are assigned $g^{x_{\text{arg}1}}$, $g^r$ and $g^{x_{\text{arg}2}} \cdot X'_{\text{output}}$, respectively. Therefore, this polynomials is set to zero iff $a = b \cdot q + r$. One should also verify that $r < x_{\text{arg}2}$. The seventh and tenth constraints are created for this reason.

The last constraint is responsible for validating the flag’s assignment. One can check that this polynomial is satisfied iff

- $x_{\text{arg}2} = 0 \rightarrow x'_{\text{flag}} = 1$
- $x_{\text{arg}2} \neq 0 \rightarrow x'_{\text{flag}} = 0$.

\textbf{umod}

\#auxiliary variables: 9$W + 1$, max degree: 4, \#polynomials: 9$W + 7

The \textit{Umod} opcode is similar to \textit{udiv}. The key difference between these opcodes is that in the former, the result should be the remainder while in the latter, it should be the quotient. One can define a $(2, 2)$ – umod opcode in the following way:

$$\mathcal{R}_{\text{umod}}(x_{\text{arg}1}, x_{\text{arg}2}, x'_{\text{output}}, x'_{\text{flag}}) =$$

$$\{x_{\text{arg}1}, x_{\text{arg}2}, 0, 1 | x_{\text{arg}2} = 0\} \cup$$

$$\{x_{\text{arg}1}, x_{\text{arg}2}, r, 0 | x_{\text{arg}2} \neq 0, x_{\text{arg}1} = q \cdot x_{\text{arg}2} + r, r < x_{\text{arg}2}, q, r \in \mathbb{N}\}$$

Due to the similarity between the \textit{udiv} and \textit{umod} opcodes, one can define a constraint system for \textit{umod} that resembles \textit{udiv}'s constraint systems. One just needs to switch the result and the remainder roles in order to get a valid constraint system for \textit{umod}.

57
5.3.4 Comparison Opcodes

Many architectures implement several integer comparison opcodes such as cmpe, cmpg, and cmpae. Usually, these comparison operations return true or false, depending on the conditional relation between the two integers. There are two major differences between these opcodes: (i) the concrete operation which is executed; for example, one opcode checks whether two integers are equal, while another opcode checks if the first integer is greater than or equal to the second; (ii) the type of the comparison — comparison of signed or unsigned integers. A sequence of bits can represent both signed and unsigned integers; i.e., 111 in unsigned representation is 7; however, in two’s complement it represents −1. Therefore, a comparison opcode indicates which kind of comparison is executed.

\( \text{cmpe} \)

**#auxiliary variables:** 1, **max degree:** 2, **#polynomials:** 2

\( \text{cmpe} \) compares two integers, and sets the flag to 1 iff the integers are equal. This (2, 1) – cmpe opcode can be described by the following relation,

\[
\mathcal{R}_{\text{cmpe}}(x_{\text{arg} 1}, x_{\text{arg} 2}, x'_{\text{flag}}) =
\begin{cases}
(x_{\text{arg} 1}, x_{\text{arg} 2}, 1) | x_{\text{arg} 1} = x_{\text{arg} 2} \\
(x_{\text{arg} 1}, x_{\text{arg} 2}, 0) | x_{\text{arg} 1} \neq x_{\text{arg} 2}
\end{cases}
\]

(5.28)

In many systems, \( \text{cmpe} \) is implemented by subtraction of the integers and comparing the result to zero. In fields with characteristic 2, addition and subtraction are the same. Hence, addition of two field elements yields 0 iff the elements are equal. The flag is set to 1 iff the addition result is 0.

\[
\mathcal{C}_{\text{cmpe}}(X_{\text{arg} 1}, X_{\text{arg} 2}, X'_{\text{flag}}, Y^1_{\text{aux}}, Y^2_{\text{aux}}) =
\begin{cases}
Y^1_{\text{aux}} + X_{\text{arg} 1} + X_{\text{arg} 2} \\
C_{\text{Indicator}}(Y^1_{\text{aux}}, X'_{\text{flag}}, Y^2_{\text{aux}})
\end{cases}
\]

The first constraint is superfluous. The above \( ACIP \) can be replaced by:

\[
\mathcal{C}_{\text{cmpe}}(X_{\text{arg} 1}, X_{\text{arg} 2}, X'_{\text{flag}}, Y_{\text{aux}}) =
\begin{cases}
(X_{\text{arg} 1} + X_{\text{arg} 2}) \cdot X'_{\text{flag}} \\
(X_{\text{arg} 1} + X_{\text{arg} 2}) \cdot Y_{\text{aux}} + 1 + X'_{\text{flag}}
\end{cases}
\]

(5.29)

One can define constraint systems for other comparison opcodes in numerous ways. For example, pursuing the \( \text{cmpe} \) idea, one can mimic an integer subtraction and compare the result to zero. Another way is by comparing the numbers bit by bit from the most significant bit (MSB) to the least significant bit (LBS); the first pair of corresponding
bits that is not equal determines the result of the opcode. In this work, we chose to present the latter option because it requires fewer constraints and is more elegant than the former option. We use the $ACIP$ defined in Equation 5.9 in order to define other comparison opcodes.

cmpa

`#auxiliary variables: 3W, max degree: 4, #polynomials: 3W + 3`

cmpa sets the flag to 1 if the first unsigned integer is above the second unsigned integer; otherwise it sets the flag to 0.

\[ C_{\text{cmpa}}(X_{\text{arg}1}, X_{\text{arg}2}, X'_{\text{flag}}, Y_{\text{arg}1}, Y_{\text{arg}2}, Y_{\text{aux}}, Y_{\text{output}}) = \]
\[ C_{\text{additive unpacking}}(X_{\text{arg}1}, Y_{\text{arg}1}) \cup \]
\[ C_{\text{additive unpacking}}(X_{\text{arg}2}, Y_{\text{arg}2}) \cup \]
\[ C_{u\text{compare}}(Y_{\text{arg}1}, Y_{\text{arg}2}, Y_{\text{output}}, Y_{\text{aux}}) \cup \]
\[ \{ X'_{\text{flag}} + Y_{\text{output}} \cdot (Y_{\text{output}} + g) \cdot (1 + g)^{-1} \} \] (5.30)

Note. Both in additive unpacking and in compare $ACIP$s, there are type checking constraints that verify that $Y_{\text{arg}1}^i$ and $Y_{\text{arg}2}^i$ are, in fact, assigned either 0 or 1. There is no need to check this condition twice; therefore, one set of this type checking constraints can be omitted.

The first three $ACIP$s have been discussed earlier. The last polynomial verifies that $X_{\text{flag}}^i$ is assigned 1 iff $y_{\text{output}} = 1$ (recall, that $Y_{\text{output}}$ can be assigned one of three values, $\{0, 1, g\}$).

cmpg

`#auxiliary variables: 3W, max degree: 4, #polynomials: 3W + 3`

The key difference between cmpg and cmpa is that in cmpg the comparison is between signed integers. In this opcode, the flag is assigned 1 iff the first signed integer is greater than the second signed integer. In practice, one should change $C_{u\text{compare}}^s$ in (5.30) to $C_{\text{compare}}^s$.

cmpae

`#auxiliary variables: 3W, max degree: 4, #polynomials: 3W + 3`

cmpae sets the flag to 1 if the first unsigned integer is above or equal to the second;
otherwise it sets the flag to 0.

\[
C_{\text{cmpae}}(X_{\text{arg1}}, X_{\text{arg2}}, X'_{\text{flag}}, \vec{V}_{\text{arg1}}, \vec{V}_{\text{arg2}}, \vec{V}_{\text{aux}}, Y_{\text{output}}) = \\
C_{\text{additive unpacking}}(X_{\text{arg1}}, \vec{V}_{\text{arg1}}) \cup \\
C_{\text{additive unpacking}}(X_{\text{arg2}}, \vec{V}_{\text{arg2}}) \cup \\
C_{u_{\text{compare}}}(\vec{Y}_{\text{arg1}}, \vec{Y}_{\text{arg2}}, Y_{\text{output}}, \vec{Y}_{\text{aux}}) \cup \\
\{X'_{\text{flag}} + Y_{\text{output}} \cdot (Y_{\text{output}} + g) \cdot (1 + g)^{-1} + \\
Y_{\text{output}} \cdot (1 + Y_{\text{output}}) \cdot (g \cdot (1 + g))^{-1}\}
\] (5.31)

If \(y_{\text{output}} = g\) or \(y_{\text{output}} = 1\), the last polynomial is satisfied when \(X'_{\text{flag}}\) is assigned 1. Otherwise, \(y_{\text{output}} = 0\), which implies that \(X'_{\text{flag}}\) should be is assigned 0.

\textbf{cmpge}

\#auxiliary variables: \(3W\), max degree: 4, #polynomials: \(3W + 3\)

As opposed to \textit{cmpae}, \textit{cmpge} compares signed integers. The flag is set to 1 iff the first signed integer is greater than or equal to the second signed integer. In order to verify that \textit{cmpge} is executed correctly, one should switch \(C_{\text{u_{\text{compare}}}}\) to \(C_{\text{s_{\text{compare}}}}\) in the ACIP presented in Equation 5.31.

### 5.3.5 Jump Opcodes

\textit{Jump} opcodes are peculiar, compared to other opcodes, in the sense that they affect only the \(PC\) register. In these type of opcodes, we assume that \(x_{\text{arg1}}\) specifies the address of the instruction being jumped to. In this section we introduce three opcodes: \textit{jmp}, \textit{cjmp} and \textit{cnjmp}. The first opcode just changes the \(PC\) to point to the address specified in \(x_{\text{arg1}}\). The other two opcodes are conditional jumps; the \(PC\)’s value is changed to the value written in \(x_{\text{arg1}}\) only if some condition is met. Otherwise, the \(PC\) is incremented by one. In \textit{cjmp}, the condition is \(\text{flag} = 1\) and in \textit{cnjmp}, the condition is \(\text{flag} = 0\) (which is the same as saying that when \(\text{flag} = 1\) the condition is not met).

\textit{Note}. Because in this section we define the \(ACIP\) for each opcode independently on other opcodes and on the program, we define type checking constraints in each opcode to force the values of the \(PC\) variables to be 0 or 1. In each \(ACIP\) for jump opcodes, there are \(\lceil \log \ell \rceil\) type checking constraints, when \(\ell\) denotes the length of the program. It is sufficient to check that the \(PC\) variables are assigned 0 or 1 only once, and not for each instance of \textit{jump} opcodes in the program.

\textbf{jmp}

\#auxiliary variables: 0, max degree: 2, #polynomials: \(\lceil \log(\ell) \rceil + 1\)

The \textit{jmp} opcode sets the \(PC\) to point to some instruction. We can describe the \textit{jmp}
opcode as a \((1, \lceil \log(\ell) \rceil) – \text{jmp opcode relation},\)

\[
\mathcal{R}_\text{jmp}(x_{\text{arg}}, x_{\text{PC}}') = \{(x_{\text{arg}}, x_{\text{arg}})\}.
\] (5.32)

This opcode’s \(ACIP\) is pretty trivial,

\[
\mathcal{C}_\text{jmp}(X_{\text{arg}}, X_{\text{PC}}') = \\
\left\{ X_{\text{PC}}^i \cdot (X_{\text{PC}}^i + 1) \mid 0 \leq i < \lceil \log(\ell) \rceil \right\} \cup \\
\left\{ X_{\text{arg}} + \sum_{i=0}^{[\log(\ell)]-1} \beta_i X_{\text{PC}}^i \right\}
\] (5.33)

where \(\ell\) is the length of the program.

**cjmp and cnjmp**

**#auxiliary variables:** 0, **max degree:** 2, **#polynomials:** \([\log(\ell)] + 2\)

In \(cjmp\), the PC is changed to the address written in the opcode if \(x_{\text{flag}} = 1\); otherwise it is increased by 1.

\[
\mathcal{R}_{cjmp}(x_{\text{arg}}, x_{\text{PC}}', x_{\text{flag}}, x_{\text{PC}}') = \\
\left\{ (x_{\text{arg}}, x_{\text{PC}}', x_{\text{flag}}, x_{\text{PC}}', x_{\text{PC}}', x_{\text{PC}} = 1, x_{\text{flag}} = 1) \right\} \cup \\
\left\{ (x_{\text{arg}}, x_{\text{PC}}', x_{\text{flag}}, x_{\text{PC}}', x_{\text{PC}} = 1, x_{\text{flag}} = 0) \right\}
\] (5.34)

Let’s assume that the cjmp opcode appears on line \(i\) in the program. We denote by \(e_i\) the field element that represents the integer \(i\). A natural way to define a constraint system for \(cnjmp\) is:

\[
\mathcal{C}_{cjmp}(X_{\text{arg}}, X_{\text{flag}}, X_{\text{PC}}') = \\
\left\{ X_{\text{PC}}^i \cdot (X_{\text{PC}}^i + 1) \mid 0 \leq i < \lceil \log(\ell) \rceil \right\} \cup \\
\left\{ X_{\text{flag}} \cdot (X_{\text{flag}} + 1) \right\} \cup \\
\left\{ X_{\text{flag}} \cdot (X_{\text{arg}} + \sum_{i=0}^{[\log(\ell)]-1} \beta_i \cdot X_{\text{PC}}^i) + (1 + X_{\text{flag}}) \cdot (e_{i+1} + \sum_{i=0}^{[\log(\ell)]-1} \beta_i \cdot X_{\text{PC}}^i) \right\}
\] (5.35)

As in the \(jmp\) opcode, \(\ell\) is the length of the program and there is no need to add type checking constraints for the PC each time \(cjmp\) is executed. It is sufficient to add these constraints only once for the whole program. Also, there is no need to check in every conditional jump that the flag variable is assigned 0, 1.

One can notice that the value of \(PC'\) is determined in the third constraint. The \(X_{\text{flag}}\) value determines the value assigned to \(PC\). If \(X_{\text{flag}}\) is assigned 1, the constraint is satisfied iff \(PC'\) is set to \(x_{\text{arg}}\). Otherwise, \(PC'\) is increased by one and set to \(e_{i+1}\).

Since \(cjmp\) and \(cnjmp\) are analogues, one can deduce \(cnjmp\) from the constraint system
5.3.6 Shift Opcodes

In shift opcodes, all numbers are treated as a sequence of bits, and the objective is to rotate the first argument by the second argument to the left (shl) or to the right (shr). In shl, Arg₂ most significant bits of Arg₁ are discarded and all the vacant bits are replaced by zeros. In shr, the opposite occurs, Arg₂ least significant bits of Arg₁ are discarded. A shift by 0 bits leaves the pattern unchanged, and a shift by more than \( W \) bits results in a zero. Hence, we can check whether there is at least one bit set on in \( W - \lceil \log W \rceil \) most significant bits of Arg₂. If so, we can automatically set the result to zero because the shift is by more than \( W \) bits. Otherwise, an actual shift should be executed. In a binary representation, a shift left is essentially equals to multiplying the number by 2, while a shift right is the same as dividing the number by 2. In the implementation of the ACIP, our objective is to mimic this behavior.

Recall, elements in \( \mathbb{F}_{2^m} \) can be represented as polynomials in \( \mathbb{F}_2[X]/I(X) \) when \( I(X) \) is an irreducible polynomial. In our implementation, we chose an irreducible polynomial such that one of the generators of the multiplicative group is the element represented by the polynomial \( X \). In order, to avoid mistakes in this section \( g \) denotes the field element represented by the polynomial \( X \). Recall, that in section 5.2 we mentioned that basis used for additive representations is the normal basis. When defining shift opcodes these assumption is particularly important because we assume that the normal basis \( \{ \beta_0, ..., \beta_{m-1} \} \) consist of consecutive powers of \( g \).

Note. In the paper [BBC⁺16], we used a different constraint system for shift opcodes, which will be presented in future work.

shl

\textbf{#auxiliary variables:} 3 \cdot W + \lceil \log W \rceil + 3, \textbf{max degree:} 2, \textbf{#polynomials:} 3W + \lceil \log W \rceil + 7

The shift left can be defined as a \((2, 2) - \text{shl} \) opcode relation,

\[
\mathcal{R}_{\text{shl}}(x_{\text{arg1}}, x_{\text{arg2}}, x'_{\text{output}}, x'_{\text{flag}}) = \\
\{(x_{\text{arg1}}, x_{\text{arg2}}, x'_{\text{output}}, (x'_{\text{output}})^{MSB} | x'_{\text{output}} = (x_{\text{arg1}} \ll x_{\text{arg2}})^{0..W-1}\} (5.36)
\]

Our objective is to build a constraint system that resembles multiplication by 2. If the number represented by Arg₂ is greater than \( W \), the result is zero. Otherwise, we want
to multiply $Arg_1$ by $g^{Arg_2}$ and extract the $W$ least significant bits.

$$C_{\text{shl}}(X_{argon}, X_{argon}, X'_{output}, X'_{flag}, Y_{argon}, Y_{isNZero}, Y_{shift}, Y_{aux}, Y_{sRes}, Y_{usRes}) =$$

$$C_{\text{additive unpacking}}(X_{argon}, Y_{argon}) \cup$$

$$\{ Y_{isNZero} + \prod_{i=1}^{W-1} (1 + Y_{argon}^i) \}$$

$$C_{\text{multiplicative packing}}(Y_{argon}, Y_{usRes}) \cup$$

$$\{ Y_{sRes} + Y_{shift} \cdot X_{argon} \} \cup$$

$$C_{\text{additive unpacking}}(Y_{sRes}, Y_{aux}) \cup$$

$$C_{\text{additive packing}}(Y_{rotation}, Y_{argon}) \cup$$

$$\{ Y_{rotation} \cdot Y_{isNZero} \} \cup$$

$$\{ Y_{rotation} \cdot Y_{inverse} + Y_{isNZero} + 1 \}$$

As discussed in regard to other ACIPs, the last polynomial is superfluous and can be omitted by replacing the variable $Y_{usRes}^{W-1}$ by $X'_{flag}$ in the polynomials. Also, the degree of the second constraint can be significantly reduced by replacing this polynomial with

$$C_{\text{additive packing}}(Y_{rotation}, Y_{argon}^{[\log W], ..., W-1}) \cup$$

$$\{ Y_{rotation} \cdot Y_{isNZero} \} \cup$$

$$\{ Y_{rotation} \cdot Y_{inverse} + Y_{isNZero} + 1 \}$$

In such a case, we added another two auxiliary variables $Y_{inverse}$ and $Y_{rotation}$, and another two polynomials, but significantly reduced the degree of the polynomials to 2 instead of $W - \lceil \log W \rceil$.

We focus on explaining the constraint system informally, because all the constraints have been proved before. The first polynomial unpacks the second argument that is responsible for the shift. The goal of the unpacking is to check whether the shift is by more than $W$ length. The fourth polynomial is basically responsible for the shift; it executes multiplication by $g^{Arg_2}$ when $Arg_2 < W$. The fifth and sixth constraints extract the $W$ least significant bits after the shift. The seventh polynomial is responsible for setting the correct result to $X'_{output}$. If $Arg_2$ is greater than $W$, the result is zero; otherwise it is set to the computed value.

**shr**

#auxiliary variables: $3 \cdot W + \lceil \log W \rceil + 3$, max degree: 2, #polynomials: $3W + \lceil \log W \rceil + 7$

In a shift right, essentially, we want to mimic the behavior of division by two. Shl resembles multiplication by 2 by multiplying the variable by $g$. The naive solution of
multiplying by $g^{-1}$ may result in a field element of a high degree, when viewed as a polynomial over $\mathbb{F}_2$. This field element may not even be the integer translation (see Section 5.2) of the integer that equals the division by 2 of the original number. In order to resolve this issue, we check if the rotation is by more than $W$. If so, the result should be zero. Otherwise, we multiply $Arg_1$ by $g^W$, and rotate the number to the right. The result of the shift appears in the $W$ most significant bits. Due to the similarity to $\text{shl ACIP}$, we leave it to the reader to complete the $\text{shr ACIP}$.

5.4 Memory validity

Recall an execution trace is memory valid if two sequential memory accesses to the same address, where the second operation is load, share the same memory values (Definition 3.2.2). We assume that a prover provides a memory sorted trace and our objective is to validate that this trace is indeed sorted according to Definition 3.2.3 and each two sequential configurations in this trace, belong to the memory relation (Equation 3.1). In order to define a constraint system for memory validity, we use an augmented configuration, which has five supplementary values,

- $\text{isMemOp}$ indicates whether a memory operation is executed. Given a configuration $C$ as stated in (3.1.2), $\text{isMemOp}$ equals 1 if $C|\Gamma_a \neq \bot$; otherwise to 0.
- $\text{isLoadOp}$ indicates if a memory load operation is executed. The $\text{isLoadOp}$ value is denoted by $\Gamma_\ell$ in a configuration Definition 3.1.2.
- $\text{Address}$ indicates the memory address accessed in the memory module, while a memory load or store operation is in progress (see page 3.1). In essence, this value is equal to $C|\Gamma_a$ when $C|\Gamma_a \neq \bot$.
- $\text{Value}$ indicates the value that is written or read from the memory module. This value is denoted by $C|\Gamma_v$ in configuration Definition 3.1.2.
- $\text{TS}$ indicates the timestep of the configuration. We use a multiplicative integer representation (see Section 5.2) to translate $\text{TS}$ into a field element. If the configuration’s timestep is $i$, the integer representation as a field element is $g^i$. The domain of possible assignments is $\{1, \ldots, g^T\}$, where $T$ is the length of a program’s execution.

**Definition 5.4.1 (Memory permutation validity).** Given a memory execution trace $S_{\text{memory sorted}} = (C'_0, \ldots, C'_T)$ that is sorted first by address and then by timestep (Definition 3.2.3), we say it is memory valid (Definition 3.2.2) and permutation valid if for each two sequential augmented configurations $C'_i, C'_{i+1}$ the following holds:

1. In $S_{\text{memory sorted}}$ all the configurations whose $\Gamma_a$ value is $\bot$ appear at the end. $C'_i|\text{isMemOp} = 0 \Rightarrow C'_{i+1}|\text{isMemOp} = 0$.

64
2. $S_{memory\_sorted}$ is sorted by memory; the $\Gamma_a$ value in $C'_{i+1}$ is greater than or equal to the $\Gamma_a$ value in $C'_{i}$. Formally,

$$C'_{i+1}|is\_Mem\_Op = 1 \Rightarrow C'_{i+1}|\Gamma_a \geq C'_{i}|\Gamma_a.$$ 

3. If two sequential configurations access the same memory address, then the latter configuration has a greater timestep. This condition validates the secondary sorting order of $S_{memory\_sorted}$. Formally,

$$C'_{i+1}|is\_Mem\_Op = 1 \land C'_{i+1}|\Gamma_a = C'_{i}|\Gamma_a \Rightarrow C'_{i+1}|TS \geq C'_{i}|TS.$$ 

4. If two sequential configurations access the same memory address and the latter configuration represents a load operation, then both configurations are requested to store or load the same value from the memory module. Formally,

$$C'_{i+1}|is\_Mem\_Op = 1 \land C'_{i+1}|\Gamma_a = C'_{i}|\Gamma_a \land C'_{i+1}|is\_Load\_Op = 1 \Rightarrow C'_{i+1}|\Gamma_v = C'_{i}|\Gamma_v.$$ 

Recall, in page 20 we discussed the first condition. We mentioned that the configuration whose $\Gamma_a$ is $\perp$ can appear either in the begging of $S_{memory\_sorted}$ or at the end. We arbitrary chose for it to appear at the end of the trace. If one wants to choose otherwise, the constraints we present further in this chapter must also be modified.

Denote $Y_{addr\_C}$ as a variable that is assigned the comparison result of two memory addresses in two consecutive configurations. In this same way, denote $Y_{TSC}$ as a variable that is assigned the comparison result of two timesteps in two consecutive configurations. Recall that the comparison of two values can result one of three values $\{0, 1, g\}$, where 0 indicates that the first number is smaller than the second, 1 indicates the first number is greater than the second, and $g$ indicates that the number are equal.

We start by presenting a constraint system that focuses on the highlights of memory permutation validity, assuming that $Y_{addr\_C}$ and $Y_{TSC}$ are set to the correct values. In order to make this system sound, we need to add more constraints such as unpacking and comparing. For example, if we want to compare $X_{Address}$ and $X'_{Address}$, we need to add the following constraints,

$$C_{additive\_unpacking}(X_{Address}, \vec{Y}_1) \cup C_{additive\_unpacking}(X'_{Address}, \vec{Y}_2) \cup C_{compare}(\vec{Y}_1, \vec{Y}_2, Y_{addr\_C}, Y_{aux})$$

In the same way, additional constraints should be added in order to verify the timestep comparison. At the end of this chapter, we present the “full” $ACIP$; see Equation 5.39. Assuming that $Y_{addr\_C}$ and $Y_{TSC}$ are assigned the values discussed above,
the ACIP we suggest for memory permutation validity is

\[ C_{\text{memory}}(X_{\text{isMemOp}}, X_{\text{Value}}, X_{\text{isLoadOp}}, X_{\text{Value}}, Y_{\text{addrC}}, Y_{TSC}) = \]
\[ \{ (1 + X_{\text{isMemOp}}) \cdot X_{\text{isMemOp}} \} \cup \]
\[ \{ X_{\text{isMemOp}} \cdot (g + Y_{\text{addrC}}) \cdot (1 + Y_{\text{addrC}}) \} \cup \]
\[ \{ X_{\text{isMemOp}} \cdot Y_{\text{addrC}} \cdot (1 + Y_{\text{addrC}}) \cdot (1 + X_{\text{isLoadOp}}) \cdot (X_{\text{Value}} + X_{\text{Value}}') \} \cup \]
\[ \{ X_{\text{isMemOp}} \cdot Y_{\text{addrC}} \cdot (1 + Y_{\text{addrC}}) \cdot (1 + Y_{TSC}) \} \]

(5.38)

**Lemma 5.4.2.** Assuming that \( Y_{\text{addrC}} \) and \( Y_{TSC} \) are assigned the correct comparison values, the constraint system (5.38) is satisfied iff (5.4.1) holds.

**Proof.** Recall that we have decided that the non memory opcodes, configurations whose \( is\text{MemOp} \) value equals zero, should appear at the end of the sorted trace. The first constraint is satisfied if either \( X_{\text{isMemOp}} \) is assigned 1 or \( X'_{\text{isMemOp}} \) is assigned 0. Meaning that the following assignments,

\[
\begin{align*}
x_{\text{isMemOp}} &= 1 \land x'_{\text{isMemOp}} = 0 \\
x_{\text{isMemOp}} &= 1 \land x'_{\text{isMemOp}} = 1 \\
x_{\text{isMemOp}} &= 0 \land x'_{\text{isMemOp}} = 0
\end{align*}
\]

satisfy the constraint as well as the material conditional,

\[ C'_{i}|\text{isMemOp} = 0 \Rightarrow C'_{i+1}|\text{isMemOp} = 0. \]

The only assignment that does not satisfy the material condition is \( x_{\text{isMemOp}} = 0 \land x'_{\text{isMemOp}} = 1 \) and it does not set the polynomial to zero.

The material condition,

\[ C'_{i+1}|\text{isMemOp} = 1 \Rightarrow C'_{i+1}|\Gamma_a \geq C'_{i}|\Gamma_a, \]

is logically equal to

\[ C'_{i+1}|\text{isMemOp} = 0 \lor C'_{i+1}|\Gamma_a \geq C'_{i}|\Gamma_a, \]

which is true if either \( x'_{\text{isMemOp}} = 0 \) or \( y_{\text{addressCompare}} \) equals 1 or \( g \). Looking at the second constraint,

\[ X'_{\text{isMemOp}} \cdot (g + Y_{\text{addrC}}) \cdot (1 + Y_{\text{addrC}}), \]

one can notice that it is satisfied if the same conditions are met.

66
The third constraint,

\[ X'_{\text{isMemOp}} \cdot Y_{\text{addrC}} \cdot (1 + Y_{\text{addrC}}) \cdot X'_{\text{isLoadOp}} \cdot (X_{\text{Value}} + X'_{\text{Value}}), \]

is satisfied iff

\[ x'_{\text{isMemOp}} = 0 \lor (y_{\text{addrC}} = 0 \lor y_{\text{addrC}} = 1) \lor x'_{\text{isLoadOp}} = 0 \lor (x_{\text{Value}} = x'_{\text{Value}}), \]

which by De-Morgan rules is equivalent to

\[ \neg(x'_{\text{isMemOp}} = 1 \land y_{\text{addrC}} = g \land x'_{\text{isLoadOp}} = 1) \lor (x_{\text{Value}} = x'_{\text{Value}}), \]

and logically equals

\[ \{ x'_{\text{isMemOp}} = 1 \land y_{\text{addrC}} = g \land x'_{\text{isLoadOp}} = 1 \} \Rightarrow (x_{\text{Value}} = x'_{\text{Value}}), \]

which matches:

\[ C'_{i+1|\text{isMemOp}} = 1 \land C'_{i+1|\Gamma_a} = C'_{i|\Gamma_a} \land C'_{i+1|\text{isLoadOp}} = 1 \Rightarrow C'_{i+1|\Gamma_v} = C'_{i|\Gamma_v}. \]

The correctness of the fourth constraint can be verified in the same manner. □

The full ACIP can be defined by:

\[
C_{\text{memory}}(X_{\text{isMemOp}}, X_{\text{Address}}, X_{\text{Value}}, X_{\text{TS}}; X'_{\text{isMemOp}}, X'_{\text{Address}}, X'_{\text{Value}}, X'_{\text{TS}}; Y_{\text{addrC}}, Y_{\text{TSC}}, Y'_{\text{Address}}, Y'_{\text{addrCompare}}, Y_{\text{TS}}, Y'_{\text{TS}}; Y'_{\text{auxTS}}, Y'_{\text{auxTS}}; Y'_{\text{TSCompare}}) = \\
\{ (1 + X_{\text{isMemOp}}) \cdot X'_{\text{isMemOp}} \} \cup \\
C_{\text{additive unpacking}}(X_{\text{Address}}, Y_{\text{Address}}) \cup \\
C_{\text{multiplicative unpacking}}(X_{\text{Address}}, Y'_{\text{Address}}) \cup \\
C_{\text{compare}}(Y'_{\text{Address}}, Y_{\text{Address}}, Y_{\text{addrC}}, Y_{\text{addrCompare}}) \cup \\
\{ X'_{\text{isMemOp}} \cdot Y_{\text{addrC}} \cdot (1 + Y_{\text{addrC}}) \cdot (X_{\text{Value}} + X'_{\text{Value}}) \} \cup \\
C_{\text{multiplicative unpacking}}(X_{\text{TS}}, Y_{\text{TS}}, Y_{\text{auxTS}}) \cup \\
C_{\text{multiplicative unpacking}}(X'_{\text{TS}}, Y'_{\text{TS}}, Y'_{\text{auxTS}}) \cup \\
C_{\text{compare}}(Y_{\text{TS}}, Y'_{\text{TS}}, Y_{\text{TS}}, Y'_{\text{TSCompare}}) \cup \\
\{ X'_{\text{isMemOp}} \cdot Y_{\text{addrC}} \cdot (1 + Y_{\text{addrC}}) \cdot (X_{\text{Value}} + X'_{\text{Value}}) \} \cup \\
\{ X'_{\text{isMemOp}} \cdot Y_{\text{addrC}} \cdot (1 + Y_{\text{addrC}}) \cdot (1 + Y_{\text{TS}}) \} \]

(5.39)
Chapter 6

Conclusion and open questions

In this work we described how to check that an instance of \((M, B)\) where \(M\) is a BT-RAM and \(B\) are boundary constraints, belongs to \(L_{CI}\). We focused on two out of four conditions in verifying that \((M, B) \in L_{CI}\), checking that an execution trace is (1) time and (2) memory valid. We defined an \(ACIP\) over a field, such that the \(ACIP\) is set to zero if and only if the execution trace is, in fact, both memory and time valid. We meticulously described the construction of the \(ACIP\) and the improvements that can be applied in order to get an succinct constraint system. The key improvements and contributions of this work are: (1) decreasing the number of variables by pre-selection; (2) algebraic arithmetization over small characteristic fields; and (3) modular construction.

While defining the \(ACIPs\), we focused on three parameters that affect verifier’s and prover’s efficiency and the PCP proof length: (i) the number of auxiliary variables; (ii) the maximum degree of the polynomials; and (iii) the number of polynomials. Each parameter can be improved at the expense of the other two. We found that for our programs, it is beneficial to reduce number of variables significantly, while increasing the degree by \(\lceil \log \ell \rceil\) where \(\ell\) is the length of the program (see the first contribution). We, however, did not show the exact trade-off among these parameters. Future research might try to find the optimal ratio among these parameters. It would be interesting to know when and by how much it is beneficial to decrease one parameter at the expense of the others.

In this thesis, we defined \(ACIPs\) for opcodes based on TinyRAM assembly language while using a linear number of auxiliary variables (see Chapter 5). TinyRAM is RAM whose arithmetic is executed over the integers. Although it is possible to define floating point arithmetic with TinyRAM opcodes, it may be inefficient. An interesting continuation of this work would be to define \(ACIPs\) for floating arithmetic, i.e., addition, subtraction, multiplication over the floating numbers. Moreover, defining \(ACIPs\) for opcodes of widely-used assembly languages such as MIPS or x86 may result in making \(PCP\) proofs more accessible. With the help of state-of-the-art compilers, it should be easy to generate assembly code for this work, which later can be translated into PCP proofs.
Bibliography


[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of


The proposal is to integrate the proposed approach with existing tools or
systems in the area of software engineering. This could be done through
integration with project management tools, code quality analysis tools, or
issue tracking systems. Such integration would allow for a more holistic
view of the software development process, enabling better management of
tasks and dependencies, and improved communication among team
members.

In conclusion, the proposed approach offers a promising solution for
managing software projects. Its potential for improving productivity and
quality while reducing costs makes it a valuable addition to the
software engineering toolkit. Further research is needed to fully
validate the proposed approach and explore its potential applications in
different domains and industries.
מכולת נישה אקריאט אלגברייה

היבר על מחקה

לשם מועיל חקקי של הורודית וק佶ית חורף
מודיע רומשבי

פרוגמאט יונגיה

חובש ליסט החברון ... מחן טכונ럭י לישראל
כסל ושערא' חפשו
דצמבר 2016
מコレקט ניסיון אקרואית אלגברית

فاءוגוס יונגסה